Universidade do Minho
Escola de Engenharia

Francisco Manuel Araújo Rocha Silva

Simulador de Movimento para Ambientes Urbanos

Outubro de 2011

Simulador de Movimento para Ambientes Urbanos
Francisco Manuel Araújo Rocha Silva
UMinho | 2011

Universidade do Minho
Escola de Engenharia

Francisco Manuel Araújo Rocha Silva

Simulador de Movimento para Ambientes Urbanos

Dissertação de Mestrado
Ciclo de Estudos Integrados Conducentes ao
Grau de Mestre em Engenharia de Comunicações

Trabalho realizado sob a orientação do
Professor Doutor Adriano Jorge Cardoso Moreira
e co-orientação da
Professora Doutora Maria João Mesquita Rodrigues da Cunha Nicolau

Outubro de 2011

É AUTORIZADA A REPRODUÇÃO PARCIAL DESTA DISSERTAÇÃO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, ___/___/_____

Assinatura: _____

“Dictionary is the only place that success comes before work.
Hard work is the price we must pay for success.
I think you can accomplish anything if you’re willing to pay the price.”
Vince Lombard

Agradecimentos

Gostaria de agradecer a todas as pessoas que, direta ou indiretamente, contribuíram para o meu sucesso a nível pessoal e académico.

Agradeço também ao Professor Doutor Adriano Moreira e à Professora Doutora Maria João Nicolau, pelas suas orientações, bem como pelo seu esforço e dedicação durante a realização deste trabalho.

Aos meus pais e madrinha, um agradecimento muito especial pelo auxílio que são no meu dia a dia e por me apoiarem sempre que foi necessário.

À minha irmã por me animar a todas as horas com o seu humor característico, pelas provocações diárias que estimularam o meu trabalho e por me lembrar que havia vida para além da dissertação.

À Joana Coelho por toda a paciência e compreensão que teve comigo, por ter sido o meu suporte nos dias em que o desânimo era mais forte, e por estar ao meu lado em todos os momentos.

A todos os que se cruzaram comigo neste percurso académico e me fizeram crescer como estudante, com especial destaque para a Joana Silva, Diogo Gomes, Diogo Mendes, Rui Ferraz e Renato Martins por terem sido os companheiros e amigos que estiveram sempre ao meu lado, pois sem eles tudo seria diferente.

Ao Sérgio Silva pela sua enorme disponibilidade sempre que precisei de ajuda com o código.

Por fim, mas não menos importante, gostaria de agradecer aos meus colegas Rui Pinheiro e Laurent Miranda, que formaram equipa de trabalho comigo neste projeto, pela entreajuda e companheirismo.

A todos, muito obrigado!

Resumo

Hoje em dia existem cada vez mais dispositivos móveis integrados na vida urbana, sendo comum encontrar-se num centro urbano pedestres e veículos transportando dispositivos com capacidades de comunicação (telemóveis, PDAs, computadores de bordo) que poderão interligar-se entre si formando um cenário de "todos ligados". Assim, torna-se importante criar novos serviços e aplicações que permitam interações diretas entre estes dispositivos de uma forma atraente tanto para os consumidores como para as operadoras.

Embora existam diversos modelos de mobilidade que tentam simular os ambientes urbanos, as conclusões a retirar sobre as interações entre nós não são suficientemente satisfatórias. Existem alguns simuladores, mais específicos, que tentam apresentar conclusões ao nível das interações, sendo que nenhum deles apresenta ainda simulações a grande escala.

É tendo como ponto de partida esta lacuna existente em simuladores de ambientes urbanos que surge o objetivo deste trabalho. Assim, pretende-se criar um simulador de mobilidade em ambientes urbanos que, além de simular as movimentações de diversos tipos de atores num ambiente urbano, permita também simular as possíveis interações entre os diversos intervenientes.

O simulador a desenvolver terá de ser capaz de simular diferentes tipos de intervenientes num ambiente urbano, como é o caso de pessoas, carros, autocarros, bicicletas, etc. Estes diferentes atores presentes no simulador terão de ter diferentes tipos de mobilidade assim como distintas reações a cada situação que surja no decorrer de uma simulação. Além de possuir uma diversa variedade de intervenientes, o simulador a implementar terá também de ser capaz de simular, recorrendo a um sistema distribuído, um número de atores o mais próximo possível da realidade urbana que se pretender simular, ou seja, terá de ser capaz de albergar numa só simulação um número muito grande de nós.

Nesta dissertação é abordado o desenvolvimento do núcleo do simulador a desenvolver, assim como a distribuição do sistema por diversas máquinas. De forma a certificar o correto funcionamento do sistema, foram efetuados testes com o objetivo de garantir que o *core* do simulador está a funcionar de forma a suportar tudo o que vai ser implementado sobre ele. Assim, foram efetuados testes de funcionalidade, de distribuição de carga, de estabilidade, de carga crítica, entre outros, sendo feito um estudo dos resultados obtidos em cada um deles.

Abstract

Today there are more and more mobile devices integrated into urban life and it's common to find in an urban center pedestrians and vehicles carrying devices with capacity of communication (mobile phones, PDAs, computers) that may be interconnected to each other forming a backdrop of 'all connected'. Thus, it is important to create new services and applications that allow direct interactions between these devices in an attractive way to both consumers and operators.

Although there are several mobility models that attempt to simulate urban environments, the conclusions to be drawn about the interactions between nodes are not sufficiently satisfactory. There are some simulators, more specific, that try to provide conclusions on the level of interactions, but none of these still has large-scale simulations.

It has as its starting point this gap in urban environments simulators appears the purpose of this work. We intend to create a mobility simulator in urban environments that, besides simulate the movements of various types of actors in an urban environment, also allows simulate about the possible interactions between different actors.

The simulator developed must be capable of simulating different types of actors in an urban environment, such as people, cars, buses, bicycles, etc. These different actors present in the simulator will have different types of mobility as well as different reactions to each situation that arises during a simulation. In addition to having a diverse range of intervenient, the simulator will also be able to simulate, using a distributed system, a number of players as close as possible to simulate the urban reality, ie, must be able to accommodate in a single simulation a large number of nodes.

This dissertation discusses the development of the simulator's core to develop, as well as the distribution system for several machines. In order to ensure the correct functioning of the system, tests were performed to ensure that the simulator's core is working to support all that is going to be implemented on it. Thus, several tests were performed such as functional tests, load distribution, stability, critical load, among others, being made a study of the results obtained in each one.

Conteúdo

Lista de figuras

Lista de tabelas

Acrónimos

API	Application Programming Interface
BartUM	Bartolomeu_Urban Mobility Simulator
CATDTN	Connectivity, Applications, and Trials of Delay Tolerant Networking
CPU	Central Processing Unit
DLR	Deutsches fur Luft- und Raumfahrt
GloMoSim	Global Mobile Information System Simulator
IDM	Intelligent-Driver Model
IP	Internet Protocol
LAN	Local Area Network
MiXim	Mixed Simulator
NS-2	Network Simulator-2
ONE	Opportunistic Network Environment Simulator
PDA	Personal Digital Assistant
SINDTN	Security Infrastructure for Delay Tolerant Network
SUMO	Simulation of Urban Mobility
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VANET	Vehicular Ad Hoc Network
WAN	Worldwide Area Network
XML	Extensible Markup Language

1 Introdução

Este capítulo tem como objetivo fazer um enquadramento do trabalho desenvolvido no âmbito desta dissertação, bem como apresentar os seus principais objetivos. Assim, este capítulo foi dividido em 3 partes. A primeira faz o enquadramento do tema e do problema abordado. Na segunda parte enumeram-se os principais objetivos a ser alcançados com este trabalho e, por fim, na terceira e última parte, é descrita a organização estrutural deste documento.

1.1 Enquadramento

Os sistemas de telecomunicações móveis são cada vez mais usados no panorama actual. Facilmente hoje se veem telemóveis, PDAs, *SmartPhones* e mesmo veículos com ligações móveis que lhes permitem comunicar entre si. Embora tenha havido uma grande evolução ao nível dos dispositivos móveis, o mesmo não se verifica com os simuladores de mobilidade que continuam a ser muito genéricos, não permitindo assim avaliar cenários específicos de mobilidade e em larga escala.

Visto que o grande mercado de dispositivos móveis está nos grandes centros urbanos, faz sentido criar um simulador que simule a dinâmica destes. Um simulador deste tipo implica o uso de diversos tipos de modelos de mobilidade adequados a este tipo de ambientes, pois num ambiente urbano podemos ter mobilidade de pedestres e de vários tipos veículos. Além disso, os ambientes urbanos possuem características muito próprias, como por exemplo, a existência de ruas, de sentidos de trânsito, de semáforos, de edifícios, de paragens de autocarros, de lojas, de cafés, etc.

Tendo em conta que este é um mercado emergente, há necessidade de se criarem simuladores capazes de simular este tipo de ambientes de forma a potenciar novos serviços e tecnologias de interação entre nós móveis.

Além da possível interação entre os dispositivos móveis, nos ambientes urbanos é fácil encontrar-se dispositivos não móveis que também se encontram ligados à rede, como

são os casos de câmaras de vigilância (seja de segurança pessoal, segurança publica ou de controlo de tráfego), sensores de temperatura e humidade, entre muitos outros.

Os simuladores existentes estão focados essencialmente nos modelos de mobilidade clássicos, esquecendo um pouco que num ambiente urbano o movimento dos atores não é sempre igual. Embora surjam cada vez mais modelos que tentam simular este tipo de comportamentos, ainda nenhum deles é efetivamente usado em simuladores específicos. Por outro lado, se o que é pretendido é simular um ambiente urbano, não se pode apenas ter em conta o tipo de movimento que os atores possam efetuar. É necessário que seja tido em conta também algumas restrições ao movimento impostas por este tipo de cenário, por exemplo, o facto de existirem muitos edifícios influencia o modo como cada nó se movimenta. Além disso, os nós não podem apresentar um movimento padrão durante a simulação, pois a realidade urbana, em que existem diversos tipos de atores, obriga a que estes tenham de ter em atenção os movimentos dos seu vizinhos, de forma a evitar colisões.

Outro factor que se deve ter em conta para simular este tipo de cenário é a existência de um número muito grande de atores numa cidade e o facto desse número estar em constante mudança. No entanto, os simuladores existentes na atualidade não preveem um grande número de atores, sendo que quando iniciam a simulação têm já um determinado número de atores que se mantém fixo durante toda a simulação.

O simulador desenvolvido tenta colmatar as lacunas apresentadas por outros simuladores, pelo que permitirá a definição de cenários com base em mapas de estradas, possibilitando ainda a colocação de edifícios. Outra particularidade é o facto da geração de tráfego pedestre e veicular ser em tempo de simulação. Este simulador suporta um grande número de atores que se movimentarão com padrões de movimento diferenciados para cada um dos tipos de atores existentes. Uma vez que a cadência de chegada e saída dos diferentes tipos de atores num ambiente urbano é diferente, é possível que os diferentes tipos de atores sejam gerados em ritmos diferentes.

Adicionalmente, pretende-se desenvolver este simulador como uma solução *open source* e multiplataforma. De forma a suportar um grande número de nós funciona como um sistema distribuído numa rede LAN dedicada ao efeito. Este simulador, desenvolvido em Java, permitirá ainda a interação entre os diversos nós que estejam relativamente próximos e a visualização do decorrer da simulação.

1.2 Objetivos

Apresentado o enquadramento em que está inserido este projeto, pode-se assim definir os seguintes objetivos gerais para a dissertação:

- Desenvolvimento de uma proposta de arquitetura para um novo simulador que seja capaz de simular a mobilidade de um número real de nós em ambientes urbanos de dimensões e características reais;
- Implementação de um protótipo funcional e de uma versão final do novo simulador, sendo capaz de simular um número muito grande de nós/atores num espaço urbano de grande dimensão;
- Fazer do simulador uma solução *open source*;
- Realizar simulações de grandes dimensões, ou seja, simulações de um grande número de nós/atores em ambientes urbanos de grande tamanho (grandes cidades), tendo como principal característica o facto de cada ator ter uma mobilidade independente, e o mais aproximado possível da realidade, assumindo todas as características dos ambientes urbanos como, por exemplo, diferenciação de sentidos de trânsito, a existência de paragens constantes devido a semáforos e passadeiras, diferentes tipos de mobilidade, locais de grande concentração de atores com um movimento reduzido, etc;

Sendo este simulador um projeto de grandes dimensões, foi constituída uma equipa de trabalho com 3 pessoas. Cada membro da equipa focou-se numa parte específica do projeto, com os seus próprios objectivos, tendo em vista a integração numa solução completa e final.

Nesta primeira fase do desenvolvimento do simulador, toda a equipa se focou nos objetivos principais do simulador tendo o trabalho a realizar sido dividido em três partes:

- Núcleo do sistema;
- Criação de diferentes atores com diferentes comportamentos em termos de movimento;
- Visualização das simulações e registo de resultados.

Esta dissertação aborda a primeira das três partes apresentadas, ou seja, o núcleo do sistema e tudo o que lhe está relacionado. Logo, os objetivos específicos correspondentes a esta dissertação são os seguintes:

- Desenvolvimento de uma proposta de arquitetura para o novo simulador que seja capaz de simular a mobilidade de um número real de nós em ambientes urbanos de dimensões e características reais;
- Elaboração de uma solução para a representação dos mapas urbanos e da configuração dos mesmos;
- Permitir o acesso remoto a toda a informação a ser usada pelo novo simulador a ser implementado, incluindo dos atores;
- Implementação, no novo simulador a ser construído, dos mapas urbanos e da sua configuração, e do acesso remoto a toda a informação a ser usada pelo novo simulador, incluindo informação referente aos atores;
- Permitir que o simulador funcione como um sistema distribuído a funcionar sobre uma rede LAN;
- Fazer a distribuição da carga do simulador de forma justa pelas diversas máquinas para que seja possível albergar o maior número de atores possíveis numa simulação;
- Construção de manuais (*web site* e documentação) de utilização e funcionamento do novo simulador, incidindo principalmente sobre os temas desenvolvidos.

1.3 Estrutura da dissertação

Esta dissertação encontra-se dividida em 6 capítulos. O primeiro capítulo trata-se de um capítulo introdutório ao tema que é abordado no decorrer da dissertação. Neste capítulo é feito um enquadramento ao tema, abordando também as motivações que levaram ao desenvolvimento do simulador, e são expostos os objetivos a atingir com a implementação do simulador, assim como os objetivos específicos desta dissertação.

No segundo capítulo, denominado de Simulação de sistemas móveis, são apresentados os temas relacionados com o tema da dissertação. Assim, são apresentados diversos modelos de mobilidade, individual e em grupo, assim como simuladores que tenham por base a mobilidade, mais concretamente em ambiente urbanos.

O terceiro capítulo deste documento, Bartolomeu_Urban Mobility Simulator (BartUM), faz a descrição do que será o simulador e o que se pretende que o mesmo seja capaz de efetuar, incidindo sobre o que foi desenvolvido no âmbito desta dissertação. É também

neste capítulo que é apresentada a arquitetura do simulador, assim como a descrição detalhada do funcionamento geral do mesmo.

O que foi desenvolvido do simulador no âmbito desta dissertação é apresentado no capítulo 4, Implementação. Aí é apresentada cada uma das classes desenvolvidas, assim como é elaborada uma explicação do funcionamento de cada uma delas. Essa apresentação das classes foi dividida por *packages* para que fosse mais fácil a associação de cada uma delas à estrutura do simulador.

No capítulo 5, Testes e Análise de resultados, como o nome sugere, são apresentados os testes efetuados para garantir o funcionamento do sistema, assim como é feita a análise dos resultados obtidos nesses testes.

Por último, o sexto capítulo, Conclusão, é onde são apresentadas as conclusões a que se chegou com o desenvolvimento deste projeto, assim como possível trabalho a executar futuramente.

2 Simulação de sistemas móveis

No presente capítulo serão apresentados aspectos relacionados com o estado atual da arte e o trabalho relacionado já existente nesta área. Para isso este capítulo foi divido em 2 secções, a primeira descreve alguns dos vários modelos de mobilidade existentes e a segunda que faz uma pequena revisão dos simuladores que existem nesta área.

2.1 Modelos de mobilidade

Uma vez que o tema central do simulador a ser desenvolvido é a mobilidade de nós em ambientes urbanos, torna-se importante abordar o tema dos modelos de mobilidade existentes. Os modelos de mobilidade que serão apresentados são os que têm maior impacto na comunidade científica atualmente.

Segundo (Camp, Boleng e Davies 2002) os modelos de mobilidade podem ser divididos em dois grupos: modelos de mobilidade individual e modelos de mobilidade em grupo. Tendo em conta o âmbito do simulador, serão mais aprofundados os modelos de mobilidade individual.

2.1.1 Modelos de mobilidade individual

Os modelos de mobilidade individual são assim designados por simularem a mobilidade de cada um dos nós da simulação individualmente (Camp, Boleng e Davies 2002). As principais características destes modelos são o facto de a velocidade e a direção serem aleatórias e não haver qualquer restrição, principalmente em relação aos restantes nós (Ribeiro e Sofia 2011). No entanto tem surgido ultimamente uns modelos que aproveitam informação temporal anterior para o movimento futuro.

Nesta secção são apresentados os modelos de mobilidade individual mais relevantes para o simulador a desenvolver.

2.1.1.1 Random Walk Mobility Model

O modelo *Random Walk Mobility*, por vezes também designado por *Brownian motion* (Camp, Boleng e Davies 2002) (Ribeiro e Sofia 2011) (Bai e Helmy s.d.), foi descrito matematicamente pela primeira vez por Einstein em 1926 (Camp, Boleng e Davies 2002), sendo por isso um dos mais antigos modelos de mobilidade a ser usado. Este modelo tem por base o movimento aleatório tanto na direção, que tem de estar compreendida entre o intervalo padrão [$0,2\pi$], como na velocidade, também ela compreendida num intervalo a ser definido [*velocidademin, velocidademax*] (Camp, Boleng e Davies 2002) (Ribeiro e Sofia 2011). Os nós móveis sujeitos a este tipo de modelo de mobilidade podem mover-se de duas formas: num movimento constante no tempo; ou num movimento constante na distância. Cada nó móvel efetua o seu movimento de um ponto inicial até a um ponto final, e só quando chega a esse ponto final é que calcula uma nova direção e uma nova velocidade. Sempre que um nó chegar a um dos limites da simulação, ele será 'refletido' para a área da simulação com um determinado ângulo em função da direção de entrada (Camp, Boleng e Davies 2002) (Ribeiro e Sofia 2011). Além de totalmente aleatório, este modelo não tem memória (*memoryless*) pois não tem em consideração qualquer caminho, velocidade ou outro tipo de informação anterior nos seus movimentos futuros (Ribeiro e Sofia 2011). Contudo, este modelo simula um movimento irreal, pois na realidade um nó, por norma, não faz mudanças de direção e velocidade abruptas como acontece com este modelo. Assim, embora seja usado em simulação de movimentos de nós, este modelo de mobilidade é mais indicado para o movimento imprevisível de partículas na física (Ribeiro e Sofia 2011) (Bai e Helmy s.d.).

2.1.1.2 Random Waypoint Mobility Model

O *Random Waypoint Mobility Model*, pela primeira vez proposto Johnson and Maltz (Bai e Helmy s.d.) (Buruhanudeen, et al. 2007), pode ser considerado uma evolução ao modelo *Random Walk Mobility*, pois também o movimento dos nós móveis é aleatório. A grande diferença entre os dois modelos de mobilidade é que no *Random Waypoint Mobility* são consideradas pausas temporais entre as mudanças de direção e/ou sentido (Ribeiro e Sofia 2011). Assim, o *Random Waypoint Mobility* pode ser descrito como um simples modelo estocástico em que os nós estão constantemente a escolher destinos e a moverem-se até eles (Aschenbruck, Gerhards-Padilla e Martini 2008). A definição do funcionamento deste modelo é simples. O nó começa por escolher um destino aleatório assim como uma veloci-

dade uniformemente distribuída no intervalo [*minvelocidade, maxvelocidade*]. Após isso, move-se para o seu destino à velocidade estipulada. Uma vez chegado ao local de destino tem um tempo de pausa, também ele uniformemente distribuído. Terminado o tempo de espera volta a procurar uma novo destino e/ou uma nova velocidade (Aschenbruck, Gerhards-Padilla e Martini 2008) (Ribeiro e Sofia 2011) (Camp, Boleng e Davies 2002). No caso de o intervalo de velocidades ser igual ao apresentado no modelo anterior ([*velocidademin, velocidademax*] = [*minvelocidade, maxvelocidade*]) e o valor temporal da pausa ser 0, então os modelos *Random Waypoint Mobility* e *Random Walk Mobility* tem um funcionamento igual (Camp, Boleng e Davies 2002). Embora este modelo tenha um funcionamento mais próximo do real, ainda apresenta várias limitações. Contudo, é dos mais usados para avaliar a performance das redes *ad hoc* devido a sua facilidade de implementação e ampla viabilidade (Bai e Helmy s.d.) (Aschenbruck, Gerhards-Padilla e Martini 2008).

2.1.1.3 Random Direction Mobility Model

Ao contrário dos modelos *Random Waypoint Mobility* e *Random Walk Mobility*, o *Random Direction Mobility* é um modelo em que os seus nós móveis tem a capacidade de se moverem até aos limites da área de simulação (Ribeiro e Sofia 2011), e foi inicialmente proposto por Royer, Melliar-Smithe e Moser (Bai e Helmy s.d.). Isto acontece numa tentativa de diminuir a densidade de nós existente no centro da área de simulação que os *Random Waypoint Mobility* e *Random Walk Mobility* podem provocar (Camp, Boleng e Davies 2002). Quando um nó é iniciado, num qualquer ponto da área de simulação, ele escolhe um ponto junto do limite dessa área e desloca-se até lá. Uma vez nesse ponto, o nó faz uma pausa temporal e escolhe uma nova direção com um ângulo compreendido entre 0 e 180º e desloca-se até esse novo ponto, perto do limite da área (Ribeiro e Sofia 2011) (Camp, Boleng e Davies 2002). Ao aplicar esse ângulo ao movimento do nó, é de prever que ele nunca saia da área de simulação (Ribeiro e Sofia 2011). Este modelo é normalmente usado quando é necessário usar-se um modelo matematicamente tratável (Musolesi e Mascolo s.d.).

2.1.1.4 Gauss-Markov Mobility Model

O modelo *Gauss-Markov Mobility* foi pela primeira vez apresentado por (Liang e Haas 1999). Neste modelo, a velocidade do nó móvel é correlacionada com o tempo e mo-

dulada como um processo estocástico Gauss-Markov (Bai e Helmy s.d.). Assim, este modelo acaba com as paragens e mudanças de direção abruptas que ocorrem nos modelos anteriores (Ribeiro e Sofia 2011) e vai adaptando a aleatoriedade dos nós a diferentes níveis através de parâmetros ajustáveis (Camp, Boleng e Davies 2002). Este modelo de mobilidade tem a particularidade de ter dependência temporal, isto significa que a velocidade e direção futuras (instante de tempo t+1) dependem dos valores atuais (instante de tempo t) (Aschenbruck, Gerhards-Padilla e Martini 2008) (Bai e Helmy s.d.) (Ribeiro e Sofia 2011) (Liang e Haas 1999). Quando um nó móvel é iniciado, a sua posição, velocidade e direção são escolhidas de forma uniformemente distribuída, sendo cada movimento feito a cada intervalo de tempo δt (Aschenbruck, Gerhards-Padilla e Martini 2008). Quando as movimentações de um nó o levam até ao limite da área de simulação, este modelo força uma mudança de direção em 180º (Bai e Helmy s.d.).

2.1.1.5 City Section Mobility Model

O modelo *City Section Mobility* foi pela primeira vez apresentado por (Jaap, Bechler e Wolf 2005). Este modelo surgiu com a necessidade de criar um modelo capaz de satisfazer as necessidades da mobilidade urbana. Assim, neste modelo os veículos movimentam-se na rede viária da cidade, onde estão incluídos os muitos cruzamentos característicos deste tipo de ambientes (Jaap, Bechler e Wolf 2005) e os diversos limites de velocidade das diferentes ruas (Camp, Boleng e Davies 2002). Desta forma, cada estrada possui um limite de velocidade e um comprimento (Ribeiro e Sofia 2011). Quando este modelo é iniciado, os nós são colocados num ponto de uma estrada e a partir desse ponto escolhe aleatoriamente outro como seu destino dirigindo-se até ele (Camp, Boleng e Davies 2002). Ao longo da sua movimentação na simulação, um veículo pode alterar a sua direção em cada cruzamento que exista no mapa, tornando o fluxo de tráfego mais heterogéneo (Jaap, Bechler e Wolf 2005). A rede de estradas que é usada neste modelo é baseada no modelo *Manhattan Grid* e todos os veículos que percorrem as ruas são iguais, não havendo diferenciação entre veículos de passageiros e camiões, por exemplo (Jaap, Bechler e Wolf 2005).

Neste modelo, a velocidade dos veículos é determinada com a ajuda do *Intelligent-Driver Model* (IDM), que permite que os diversos nós ajustem a sua velocidade em função dos que vão à sua frente (Jaap, Bechler e Wolf 2005). Pelas suas características, o modelo

City Section Mobility é mais utilizados em redes veiculares Ad-Hoc (VANETs) (Ribeiro e Sofia 2011) (Jaap, Bechler e Wolf 2005).

2.1.1.6 Manhattan Grid Mobility Model

A primeira vez que o modelo *Manhattan Grid Mobility* foi apresentado, por (Bai, Sadagopan e Helmy 2003), tinha o objetivo de emular um movimento padronizado de nós móveis em mapas de ruas. Os mapas que são usados por este modelo são compostos apenas por ruas horizontais e verticais, tendo cada uma delas dois sentidos, Norte e Sul para as verticais e Este e Oeste para as horizontais (Bai, Sadagopan e Helmy 2003). Inicialmente os nós móveis são colocados num local aleatório do mapa (Ribeiro e Sofia 2011) (Aschenbruck, Gerhards-Padilla e Martini 2008) e vão deslocando-se ao longo das redes verticais e horizontais (Bai, Sadagopan e Helmy 2003). Sempre que um nó chega a um cruzamento ele pode optar por virar à esquerda, virar à direita ou seguir em frente. A sua opção por um dos caminhos é baseada em probabilidades, sendo a probabilidade de seguir em frente de 0,5 e a probabilidade de virar para um dos lados, esquerda ou direita, de 0,25 para cada um deles (Bai, Sadagopan e Helmy 2003).

No que diz respeito à velocidade dos nós, ela é dependente do instante de tempo anterior e também do nó que o procede na sua rua e no seu sentido (Bai, Sadagopan e Helmy 2003). Assim, pode-se afirmar que este modelo tem uma dependência espacial e uma dependência temporal (Bai, Sadagopan e Helmy 2003).

2.1.2 Modelos de mobilidade em grupo

Os modelos de mobilidade em grupo são todos os modelos de mobilidade em que as movimentações dos nós móveis da simulação são dependentes entre si (Camp, Boleng e Davies 2002). Assim, estes modelos têm como padrão de movimento todo um grupo de nós (Ribeiro e Sofia 2011) e não uma movimentação individual.

Segundo (Hong, et al. 1999), uma dos primeiros modelos de mobilidade em grupo foi apresentado por (Bergamo, et al. 1996) e era denominado por *Exponential Correlated Random Mobility*. Neste modelo, o movimento dos grupos de nós tem em consideração a posição atual para o cálculo da sua nova posição (Camp, Boleng e Davies 2002), mesmo sendo a movimentação aleatória.

No entanto, em (Hong, et al. 1999) é também apresentado um modelo de mobilidade denominado de *Reference Point Group Mobility*. Neste modelo, cada grupo de nós possui um 'centro' lógico que define o comportamento desse grupo, incluindo o movimento, a velocidade, a direção, a aceleração, etc. Assim, os nós de cada grupo tem movimentações individuais mas restritas aquele ponto de referência.

Um outro modelo de mobilidade em grupo usado é o *Column Mobility* proposto por (Sánchez e Manzoni 2001). Neste modelo os nós móveis movimentam-se em grupo em torno de uma linha ou coluna, sendo os nós apenas andam em frente (Camp, Boleng e Davies 2002).

O modelo *Nomadic Community Mobility*, também ele proposto por (Sánchez e Manzoni 2001), tem como base a populações nómadas e as suas movimentações. Assim, os nós movimentam-se de tempos a tempos em grupo, sendo que no restante tempo os nós se podem movimentar individualmente dentro da área do grupo.

Em (Sánchez e Manzoni 2001) é apresentado também o modelo *Pursue Mobility*. Este modelo tem como funcionamento a perseguição de um nó, ou seja, existe um nó móvel que se vai movendo ao longo do mapa definido e os restantes seguem-no numa espécie de excursão atrás de um guia.

Por fim, o modelo *Community Based Mobility*, reinventado em (Musolesi e Mascolo 2006), agrega um grupo de nós móveis de comportamento semelhante formando comunidades. O movimentos dos nós, baseado em probabilidades, é que vai fazendo que eles se associem às diversas comunidades existentes.

2.2 Simuladores existentes

A quando do estudo inicial sobre os simuladores existentes, como a equipa era composta por 3 elementos, foram atacadas 3 áreas distintas de simuladores. Assim, as áreas que se decidiu explorar foram:

- Mobilidade urbana;
- Modelos de mobilidade;
- Mobilidade veicular;

A escolha de abordagem destas áreas de estudo foi definida tendo por base as possíveis necessidades que irão existir na planificação e implementação do sistema pretendido.

Isto acontece porque o simulador a desenvolver estará inserido nestas três áreas, sendo por isso de relevante importância conhecer bem que existe nestes campos.

Em cada uma das áreas apresentadas foi pesquisado qual o simulador que preenchesse requisitos semelhantes aos que pretendias-mos para o nosso simulador de forma explorar o que já existe ao nível dos simuladores e também para poder tentar melhorar o que já existe nessa vertente. Um factor decisivo na escolha dos simuladores a estudar era o facto de serem, tal como o nosso, *open source*.

Assim, os simuladores escolhidos foram:

- Mobilidade urbana – *The ONE* (The ONE 2010);
- Modelos de mobilidade – *BonnMotion* (Informatik 4:BonnMotion 2010);
- Mobilidade veicular – *SUMO* (SUMO - Simulation of Urban Mobility 2010);

Aquele que será mais exaustivamente descrito nesta dissertação é o *The ONE*, pois a área destinada a explorada foi a mobilidade urbana.

2.2.1 The ONE

O The ONE (*Opportunistic Network Environment*) é, como o nome indica, um simulador de redes oportunísticas. Este simulador está a ser desenvolvido pelos projetos SINDTN (*Security Infrastructure for Delay Tolerant Network*) e CATDTN (*Connectivity, Applications, and Trials of Delay Tolerant Networking*) e conta com o apoio da *Nokia Research Center* (*Finland*). O simulador está a ser desenvolvido em Java e é um projeto *open source*, estando o seu código disponível para *donwload* no site do projeto (The ONE 2010). Ao nível da portabilidade, sendo o simulador uma aplicação em Java, o The ONE é bastante flexível uma vez que funciona sem problemas nos principais sistemas operativos da atualidade, *Windows, Mac OSX* e *Linux*.

Este simulador é capaz de suportar nós móveis com diferentes modelos de mobilidade, troca de mensagens entre os nós, vários algoritmos de *routing* entre emissores e receptores e ainda possui uma interface gráfica que permite, entre outras coisas, a visualização da movimentação dos nós e da troca de mensagens entre os mesmos em tempo-real (Keränen, Ott e Kärkkäinen 2009).

2.2.1.1 Ambiente gráfico

Quando iniciado, este simulador mostra o seu ambiente gráfico, uma janela onde se pode ver e ter acesso aos nós que estão ativos (*Nodes*), as ligações que os nós vão efetuando (*Event log*), uma representação gráfica dos atores presentes na simulação, assim como os caminhos que eles vão seguindo.

Na Figura 2.1 pode ver-se o ambiente gráfico padrão deste simulador. Além da visualização da movimentação dos nós no mapa, há ainda outras funcionalidades, disponíveis numa barra acima da visualização que permite, por exemplo, captar uma imagens de um determinado momento, possível através do botão *screen shot*, efetuar pausas na simulação, fazer a simulação correr mais depressa ou devagar, ou ainda, a possibilidade de se fazer *zoom* ao mapa apresentado no ecrã.

Na parte inferior do ambiente gráfico, na zona *Event log*, é possível observar as mensagens que vão sendo trocadas entre os vários nós. O tipo de mensagens que vai surgindo na zona *Event log* é escolhido através do menu existente no seu lado esquerdo, *Event log controls*.

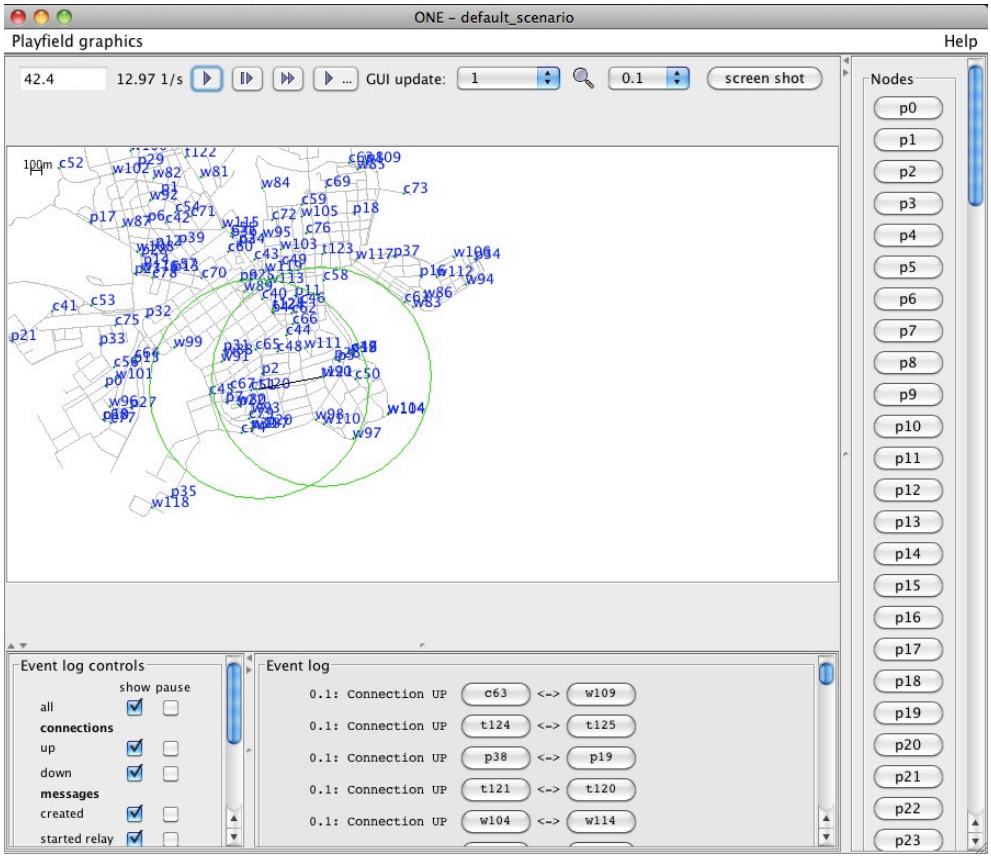

Figura 2.1 - Ambiente Gráfico do The ONE

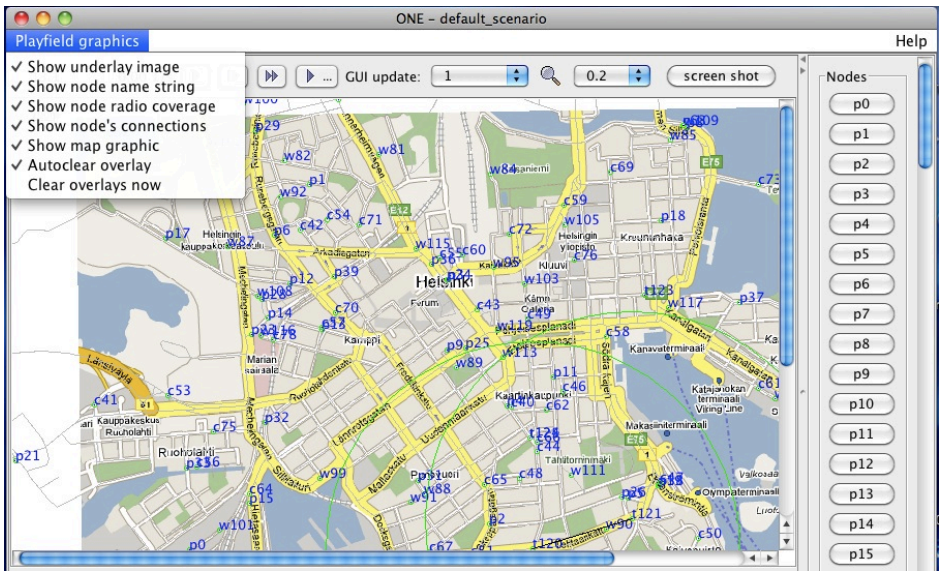

Figura 2.2 - Ambiente Gráfico com o mapa como fundo

Como se pode ver na Figura 2.2, no canto superior esquerdo do ecrã aparece um menu com o nome de *Playfield graphics*. Se se abrir esse menu aparecem várias opções de visualização na janela que contém os nós e os seus caminhos, como é o caso do mapa base que foi usado para construir o percurso apresentado, neste caso um mapa do centro de Helsínquia.

2.2.1.2 Cenários de simulação

No friso superior da janela do simulador é possível ver-se o nome da simulação que está a decorrer. Tanto na Figura 2.1 como na Figura 2.2 o cenário que está a ser simulado tem o nome de *default_scenario*. Trata-se do cenário que é aberto por defeito quando não é especificado qualquer tipo de cenário quando se arranca o software. Para se iniciar um cenário diferente basta colocar o nome do cenário que se pretende arrancar após o comando para iniciar o simulador.

Caso se pretenda simular um cenário específico, este tem que ser configurado através de um conjunto de parâmetros descritos num ficheiro de texto, e tem que ser escrito segundo as regras dos ficheiros *properties* do Java.

Quando fazemos o *download* do pacote *one_1.4.0.tar.gz.gz* já vêm incluídos alguns cenários de teste além do *default_scenario*, que permitem executar simulações tendo por base alguns dos modelos de mobilidade apresentados anteriormente.

Se se pretender explorar diferentes tipos de cenários para além dos que o The ONE fornece, é aconselhado consultar o cenário *default_settings.txt* e seguir as instruções que estão lá disponibilizadas sobre cada uma das primitivas usadas e sobre os tipos de primitivas que se podem acrescentar. Outro ficheiro que pode ajudar a entender utilidade de algumas das linhas dos ficheiros de cenário é o *WDM_conf_help.txt*.

2.2.1.3 Novos mapas

Para a realização de um novo mapa de estradas é necessário recorrer a uma outra aplicação, também ela *open source*, que cria um mapa vectorial num formato muito específico, *.wkt*. O nome dessa aplicação é *OpenJUMP* (OpenJUMP GIS) e já conta com a versão 1.4.0.2 para os sistemas operativos mais usuais (*Windows, Mac OSX, Linux*). Este *software* possibilita a criação de um mapa vectorial, tendo por base uma imagem do mapa que se deseja criar. Desta forma é mais simples a criação desses mapas com uma aparência muito próxima do real.

De forma a ser possível ter no simulador vários grupos diferentes a agirem de forma também ela diferente sobre o mesmo caminho, convém serem criados diferentes mapas sobre o mesmo mapa. Para isso o *OpenJUMP*, como um qualquer editor de imagem vectorial, permite a criação de várias *layers* sobre o mesmo projeto.

Como exemplo, pode-se imaginar uma cidade que tenha uma zona pedonal. Na fase inicial faz-se uma *layer* com uma imagem de todas as ruas da cidade, sejam elas pedonais, destinadas só a transportes públicos, só a veículos ou a todos eles. Posteriormente, podem ser criadas *layers* em que são especificados os caminhos, como o caso dos caminhos pedonais, sendo que cada *layers* que for criada pode ser guardada como um mapa. Se se quiser, por exemplo, permitir que diferentes tipos de nós usem o mesmo percurso, será vantajoso que se crie diferentes *layers*, ou seja, diferentes mapas, para cada um dos tipos de nós.

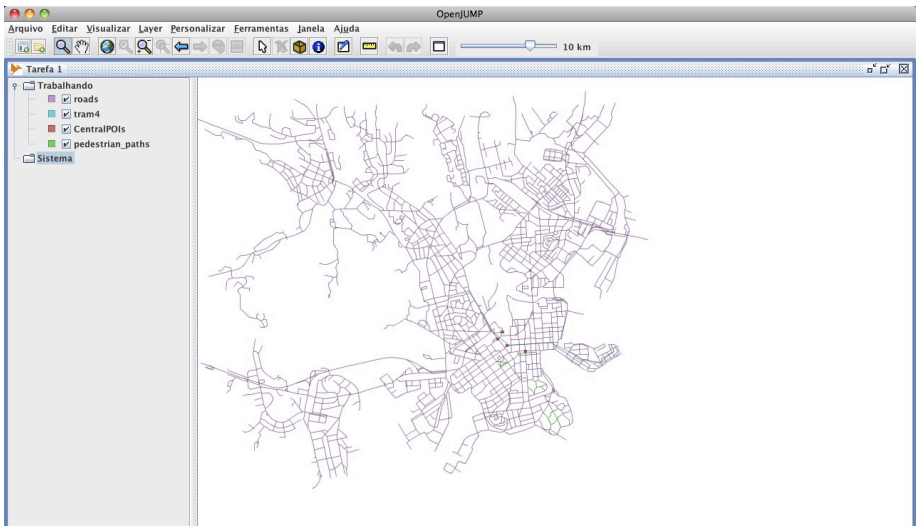

Figura 2.3 - Ambiente Gráfico do *OpenJUMP*

2.2.1.4 Vantagens

O *The ONE* é um simulador que permite simular o movimento de diversos tipos de nós assim como as suas interações. Este simulador tem um ambiente gráfico bastante intuitivo e vai mostrando ao utilizador, em tempo real, quando é que os nós se conectam e desconectam. Além disso, permite que o utilizador controle que tipo de *logs* que pretende ver durante a simulação, sendo que os pode alterar a qualquer momento (Figura 2.1). Uma outra característica que é muito relevante no *The ONE* é o facto de se poder acelerar ou abrandar a velocidade da simulação. De forma a que seja visualmente mais atrativo, o *The ONE* dispõe da possibilidade de se colocar uma imagem da cidade que está a simular como fundo no visualizador (Figura 2.2). Por fim, podemos ainda referir que este simulador é multiplataforma e permite que sejam carregados diferentes mapas.

2.2.1.5 Limitações

Embora o *The ONE* seja um simulador com muito bons atributos possui algumas limitações que estão constantemente a ser melhoradas, visto ser um projeto ainda em desenvolvimento. As principais limitações encontradas foram:

- a quantidade de atores é sempre a mesma durante toda a simulação;
- os atores não são dotados de inteligência ao nível do seu movimento, ou seja, fazem sempre o mesmo circuito com a mesma velocidade;
- apesar dos atores se conectarem, aparentemente não trocam qualquer tipo de informação.

2.2.2 BonnMotion

O *BonnMotin* é uma aplicação Java utilizada para criar e analisar cenários de mobilidade. Desenvolvido pelo grupo de Sistemas de Comunicação da Universidade de Bonn, na Alemanha, serve de ferramenta na investigação de dispositivos móveis.

Este simulador permite criar vários tipos de cenários de mobilidade que seguem as regras dos modelos de mobilidade a eles associados, tendo sido alguns deles explicados anteriormente. Os modelos de mobilidade suportados incluem, entre outros, os seguintes:

- *Randon Waypoint Mobility;*
- *Manhttan Grid Mobility;*
- *Gauss-Markov Mobility (original);*
- *Gauss-Markov Mobility;*
- *Reference Point Group Mobility;*
- *Randon Direction Mobility;*
- *Randon Walk Mobility;*
- *Column Mobility;*
- *Nomadic Community Mobility;*

O formato nativo em que *BonnMotion* guarda as trajetórias do movimento dos pontos de passagem do nó é uma linha no ficheiro, ou seja, cada linha do ficheiro corresponde a todas as movimentações de um determinado nó, o que significa que existe uma linha para cada nó. Esta linha contém todos os pontos por onde o nó passou. Os cenários criados por este simulador podem ser exportados para outros simuladores de redes para que sejam usados por esses mesmos simuladores, nomeadamente pelos seguintes:

- NS-2 (The Network Simulator - ns-2 2010);
- GloMoSim/QualNet (GloMoSim 2010);
- COOJA (An Introduction to Cooja 2010);
- MiXiM (MiXiM 2010);
- The ONE (The ONE 2010);

Além destes formatos, o *BonnMotion* permite ainda que se guarde a informação em XML.

Esta aplicação não possui qualquer tipo de ambiente gráfico para acompanhar ou analisar a simulação. No entanto, existe a possibilidade de se exportar um ficheiro que permite seguir a movimentação de um dos nós usando o *gnuplot*.

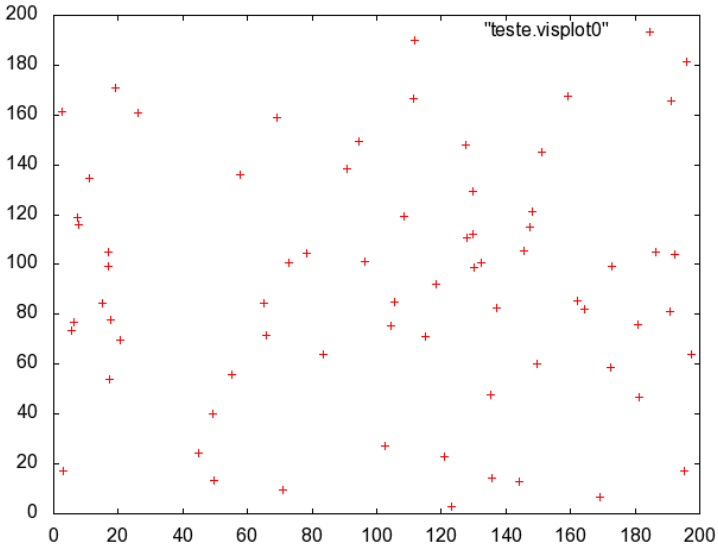

Figura 2.4 - Exemplo de um cenário de teste visto no *gnuplot*

Sendo um *software open source*, o *BonnMotion* encontra-se actualmente na versão 1.5a lançada a 03-03-2011 e está disponível para *download* no *site* do projecto (Informatik 4:BonnMotion).

2.2.3 SUMO

O SUMO (*Simulation of Urban MObility*) é uma aplicação, *open source*, de simulação de tráfego multi-modal, desenvolvido em linguagem C++. Este projecto, iniciado em 2000, conta com um vasto grupo de participantes composto maioritariamente por universidades alemãs sendo elas: Zaik – University of Cologne; DLR (Deutsches fur Luft und Raumfahrt); University of Lubeck; Humboldt University of Berlin; University of Innsbruck; Technical University of Munich; Indian Institute of Technology Bombay; Polytechnic of Torino; e University of Wroclaw.

Esta aplicação tem como objectivo simular trânsito urbano, dando a possibilidade de criar mapas e viaturas de vários tipos, sendo que também se pode controlar o fluxo de trânsito a gerar.

Para se criar uma simulação é necessário criar quatro ficheiros XML, um para definir pontos de estrada (extensão *.nod.xml*), um para definir os sentidos de circulação (exten-

são *.edg.xml*), um outro que define os percursos pretendidos assim como as viaturas (extensão *.rou.xml*) e por fim um ficheiro gerado pelo comando "NETCONVERT" (extensão *.net.xml*). Este último importa os ficheiros onde estão as estradas e os sentidos de circulação e retorna um outro ficheiro onde se pode controlar os limites de velocidade e o tipo de veículos que irão circular durante a simulação.

Para efectivamente correr um exemplo de simulação, é necessário criar um outro ficheiro, com a extensão *.cfg*, onde indicamos os ficheiros *.net* e *.rou* para o SUMO saber o que executar. É também neste ficheiro que se define a duração da simulação.

Após a criação dos ficheiros é possível executar o programa e é carregado o ficheiro criado, de forma a dar inicio à simulação.

3 Bartolomeu_Urban Mobility Simulator (BartUM)

Após terem sido estudados os trabalhos que, de diversas formas, estavam relacionados com o projeto a efetuar, avançou-se para realização da análise de requisitos e a descrição do simulador a desenvolver de forma a ser construída uma base estruturada e sólida.

Assim, neste capítulo será apresentada a análise de requisitos feita, assim como uma descrição de como o simulador foi pensado e desenvolvido.

3.1 Objetivos

O desafio proposto com este trabalho foi a elaboração de um simulador capaz de simular a mobilidade em ambientes urbanos. Os objetivos principais do simulador a desenvolver são:

- Simular mobilidade em ambientes urbanos em larga escala (grande áreas, muitos nós, etc.);
- Simular redes móveis inseridas nesses mesmos ambientes.

3.2 Princípios fundamentais da abordagem

De forma a elaborar uma abordagem correta ao simulador a desenvolver, foi definida uma lista de princípios fundamentais para servirem de orientação na elaboração a sua arquitetura, sendo eles:

- Cada entidade (pessoa, veículo, *tram*, etc.) será representada por um ator autónomo;
- Cada um dos atores será gerado em tempo de simulação, existindo geradores específicos para cada tipo de entidade, e seguindo modelos de geração que repre-

sentem uma dada realidade urbana (ex: seguindo um modelo de entrada de veículos numa cidade através de numa dada rua);

- A mobilidade dos atores será baseada em modelos comportamentais (andarão sobre estradas, considerarão o comportamento dos outros atores no seu movimento futuro, etc.), um pouco como acontece no modelo de mobilidade *City Section* (apresentado na secção 2.1.1.5);
- O espaço de simulação será baseado em mapas, que representam as ruas, as linhas de metro ou *tram*, os edifícios, etc.;
- Cada ator será representado, na simulação, por um processo independente (*threads*);
- Os atores partilharão um espaço em memória para suportar as interações entre eles (Figura 3.1);

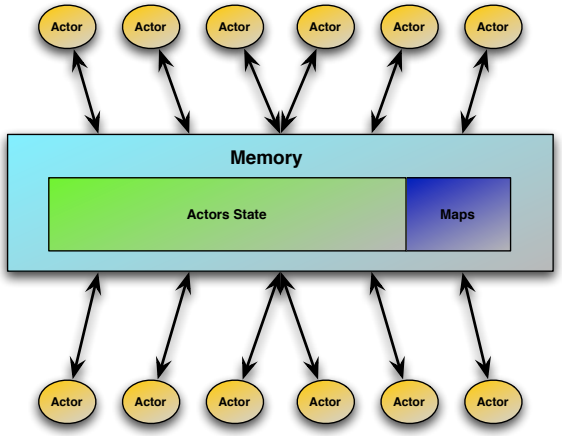

Figura 3.1 - Memória partilhada por vários atores

- Tendo em conta que se pretendem simular situações que envolvem grandes áreas e um elevado número de atores, estabeleceu-se que a solução representada na Figura 3.1 seria distribuída por um conjunto de computadores, em número arbitrário, tal como é representado na Figura 3.2;

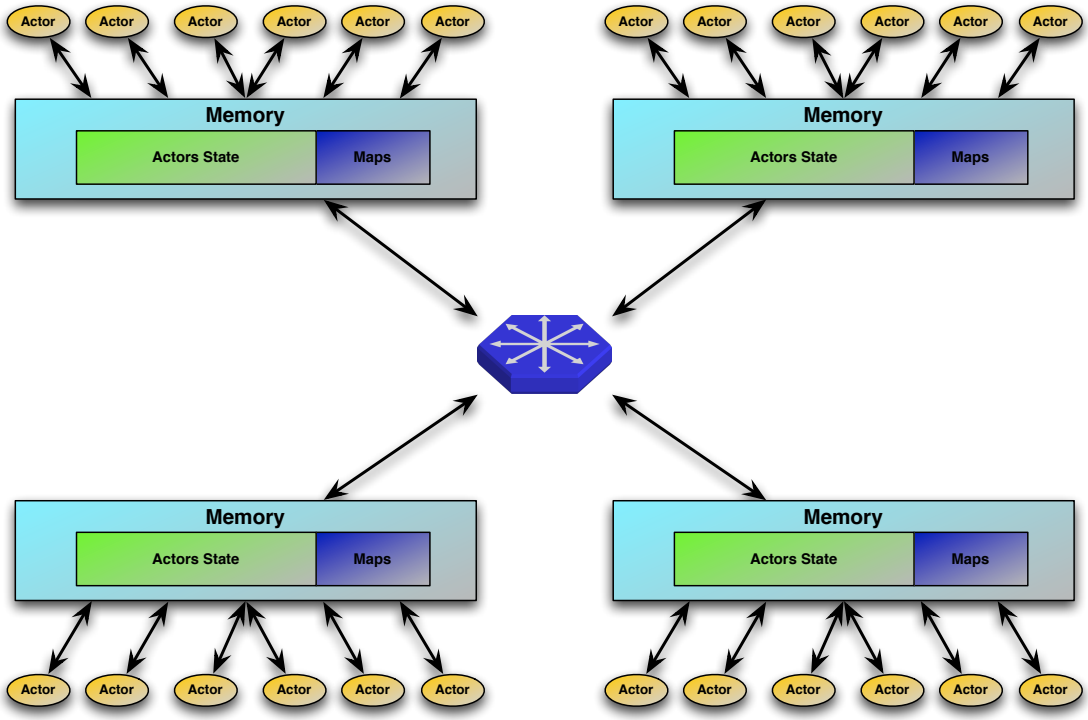

Figura 3.2 - Sincronização da memória partilhada

- Para suportar a distribuição de carga por vários computadores, é necessário adicionar à solução base (Figura 3.1) um processo que mantenha a memória partilhada sincronizada;
- Tendo em conta que os atores são gerados em tempo de simulação, e no sentido de garantir uma distribuição uniforme da carga computacional pelos vários computadores, introduziu-se o conceito de *Coordinator* (Figura 3.3). Este *Coordinator* deverá assumir a responsabilidade pela geração dos atores, através das entidades *Generator*, e pela sua distribuição pelos vários computadores disponíveis;

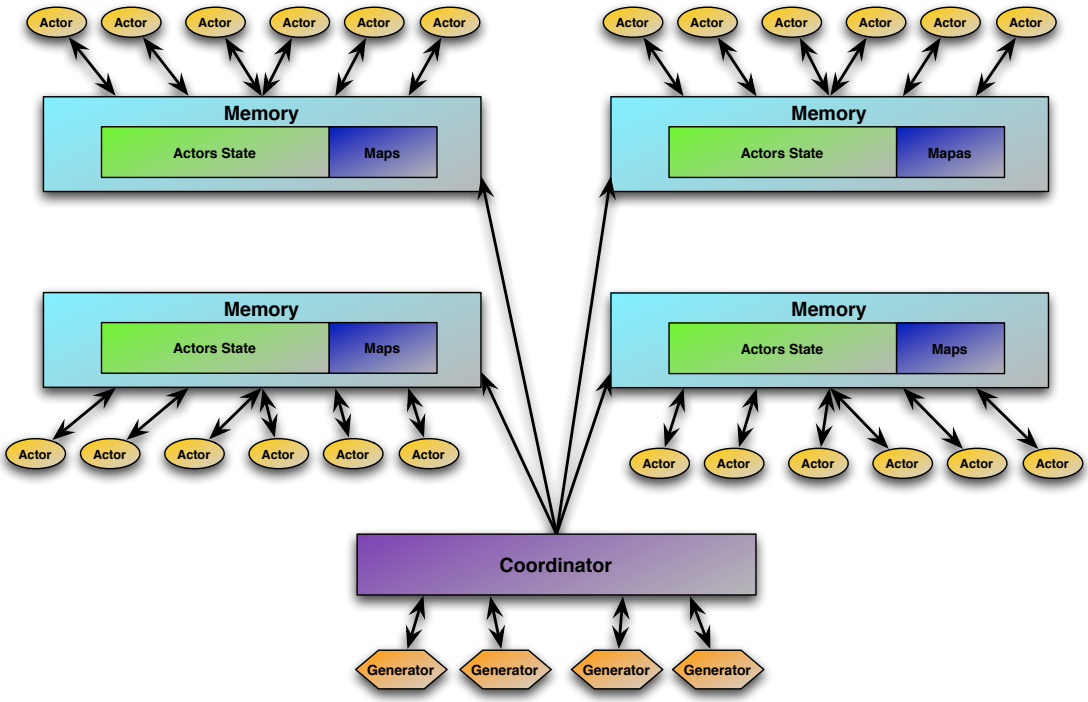

Figura 3.3 - Distribuição de carga por diversas máquinas

3.3 Arquitetura do sistema

Analisados os princípios fundamentais ao desenvolvimento do BartUM Simulator (Bartolomeu Urban Mobility Simulator) passou-se a desenhar uma arquitetura que fosse capaz de suportar esses princípios.

A base de concepção da distribuição da arquitetura foi pretender-se implementar um sistema que repartisse a carga por diversas máquinas de forma a ser possível albergar o maior número de atores possível numa só simulação. Essa arquitetura está representada na Figura 3.4.

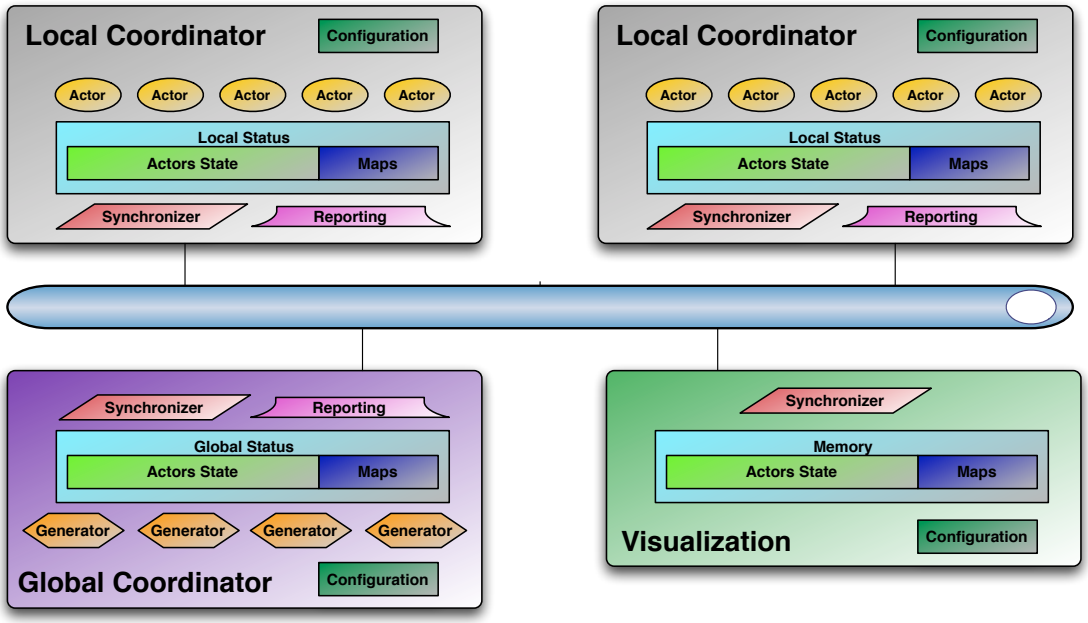

Figura 3.4 - Arquitetura do sistema

Na Figura 3.4 podem-se identificar 3 entidades nucleares para o funcionamento do sistema, e são elas o *GlobalCoordinator*, o *LocalCoordinator* e a *Visualization*. Estas entidades terão um funcionamento autónomo, embora necessitem de estar em permanente contacto. Cada uma destas entidades é executada num computador diferente, estando estes ligados através de uma rede de área local (LAN).

A primeira entidade, o *GlobalCoordinator*, funcionará como coordenador do desenrolar da simulação. Por esse motivo, no sistema a desenvolver só haverá um *GlobalCoordinator* por simulação. Esta entidade é composta por outras quatro entidades fundamentais: o *GlobalStatus*, o *Reporting*, o *Synchronizer* e os *Generators*, sendo que existirão várias destas entidades no *GlobalCoordinator*. O *GlobalStatus* é um espaço de armazenamento, e mantém o estado de cada ator, bem como todos os mapas que representam o espaço físico a simular. O *Synchronizer* é o processo responsável pela sincronização dos espaços de memória partilhados por todos os atores em todos os computadores. Os *Generators* são responsáveis pela geração de novos atores. O *Reporting* tem por função registar a evolução da simulação ao longo do tempo, para posterior análise.

O *LocalCoordinator*, que terá por base o *GlobalCoordinator*, funcionará como um coordenador local que terá atores associados a si e que assim sendo os vai monitorizando. Esta entidade segue a base do *GlobalCoordinator*, onde também existem quatro entidades fundamentais: o *LocalStatus*, com configurações semelhantes às do *GlobalStatus*, o *Syncro-*

nizer, o *Reporting* e os *Actores*. Estes últimos representam cada uma das entidades a simular.

Por último a entidade *Visualization*, tem por função permitir observar, em qualquer momento, o estado da simulação através de uma vista gráfica dos mapas e dos atores. Esta entidade não influencia a simulação em si, e pode ser ativada apenas em alguns instantes para efeitos de monitorização.

Como é perceptível na Figura 3.4, todas as entidades possuem a entidade *Configuration*. Esta entidade consiste em um ficheiro, do tipo *properties*, que é carregado no início de cada simulação. Este ficheiro contém informação necessária à inicialização de cada uma das entidades. Um exemplo do que pode conter um ficheiro deste tipo são as informações sobres os mapas a carregar pelo *GlobalCoordinator*.

3.4 Componentes do sistema

De forma a possibilitar o alcance dos objetivos propostos, foram então definidas 8 entidades que se vão relacionar entre si, sendo elas: *GlobalCoordinator*, *GlobalStatus*, *LocalCoordinator*, *LocalStatus*, *Visualization*, *Generator*, *Reporting* e *Actor*.

3.4.1 GlobalCoordinator

O *GlobalCoordinator* será o coordenador do desenrolar da simulação. É ele quem decidirá quando um novo ator é criado e a que *LocalCoordinator* ficará associado, de forma a que a carga seja distribuída por todas as máquinas. É também esta entidade que assume as funções de sincronização da memória partilhada.

3.4.2 GlobalStatus

Esta entidade guardará toda informação relativa ao estado atual da simulação, sendo que, por essa razão, vai constantemente receber informações enviadas pelos *LocalCoordinators* de maneira a que a sua informação se mantenha atualizada. É também nesta entidade que se encontrará a informação relativa aos mapas, necessários à *Visualization* e aos *LocalCoordinators*, que os poderão receber pedindo ao *GlobalCoordinator* que este lhes envie a informação de que necessitam.

3.4.3 LocalCoordinator

Como definido na secção 3.2, o sistema a construir irá ser distribuído, levando à necessidade de se criar entidades que mantenham o sistema sempre atual para todos os computadores envolvidos, ou seja, sincronizar a memória partilhada. Surgiu assim o conceito de *LocalCoordinator*, que é bastante semelhante ao *GlobalCoordinator*. A diferença mais assinalável entre o *GlobalCoordinator* e o *LocalCoordinator* será o facto de este último ter atores associados a si, ao contrário do outro. Assim, o *LocalCoordinator* será um coordenador do estado local, baseado no *GlobalCoordinator*, sendo que este, periodicamente, terá que enviar o estado dos seus atores para que a memória partilhada esteja sempre atualizada.

3.4.4 LocalStatus

Tal como acontece com o *GlobalCoordinator* e o *LocalCoordinator*, também o *LocalStatus* é muito semelhante ao *GlobalStatus*. Esta entidade também guardará toda informação relativa ao ponto atual da simulação. No entanto ela diferenciará os atores que estão no mesmo *LocalCoordinator* dos restantes, para que o *LocalCoordinator* consiga saber qual a informação que necessita de enviar para a rede. Esta entidade guardará ainda informação relativa aos mapas necessários aos atores do seu *LocalCoordinator*.

3.4.5 Visualization

A entidade *Visualization* é responsável por criar um *applet* que fará a apresentação gráfica da forma como a simulação está a decorrer, ou seja, representa o movimento que os atores estão a ter. Para que seja uma representação mais perceptível, a *Visualization* necessitará de ter todos os mapas carregados para que os possa representar. Além disso, terá que ser constantemente informada das movimentações dos atores para que consiga representar o seu movimento. Esta entidade não é obrigatória, ou seja, não tem que ser criada no início da simulação, assim como não precisa de estar sempre ativa, podendo ser ligada apenas quando se achar oportuno monitorizar o estado de uma simulação, ou até mesmo nunca ser ligada. A *Visualization* será responsável por uma interação mais direta entre a simulação e o utilizador.

3.4.6 Actors

Os *Actors* serão as entidades mais importantes e mais elaboradas do sistema a desenvolver, pois eles terão que estar em constante movimento. Esta entidade será independente, podendo ser fixa ou móvel, sendo que é um interveniente no curso da simulação. Os atores podem representar um *smartphone* na posse de uma pessoa, um veículo, um semáforo, ou seja, um qualquer dispositivo que seja passível de se conectar a outro. Assim, o movimento não será feito de forma totalmente aleatória, sendo um dos objetivos que os atores sejam o mais próximo possível de um ator real. Assim, eles terão de ser suficientemente desenvolvidos ao ponto de saberem que existe alguém à sua frente, que uma determinada estrada tem apenas um sentido, que existem cruzamentos e semáforos, que há pontos de encontro onde eles poderão parar, etc. Cada *Actor* que for criado tem de ficar associado a um dos *LocalCoordinators* existentes e terá que lhe comunicar qualquer tipo de alteração que surja no seu estado, para que a informação guardada no *LocalStatus* nunca fique obsoleta. Por forma a que os vários nós se possam movimentar ao mesmo tempo sem que haja problemas, cada um correrá numa *thread* diferente. Assim, será possível ter vários *Actors* a movimentarem-se ao mesmo tempo.

Nesta fase de implementação, foram pensados 6 tipos diferentes de atores, *tram*, *pedestrian*, *car*, *train*, *cycle* e *bus*.

3.4.7 Generators

As entidades *Generators* serão responsáveis pela criação dos novos *Actors*. Cada *Generator* será especializado em apenas um tipo de *Actors* que serão colocados num determinado ponto fixo do mapa para começarem o seu movimento a partir daí. A geração de *Actor* não será feita com uma cadência igual para todos os tipos de *Actors*, pois é normal que num ambiente urbano exista maior oscilação do número de pedestres do que em relação ao número de comboios ou autocarros, ou seja, serão usados diferentes modelos para cada tipo de ator. Esse intervalo de geração de novos atores irá ocorrer segundo um determinado padrão temporal e probabilístico.

3.4.8 Reporting

A entidade *Reporting* será a responsável por fazer o relatório de tudo o que se vai passar na simulação. Para que seja possível fazer uma correta análise e diagnósticos sobre o

que se passará na simulação esta classe terá que estar presente no *GlobalCoordinator*, pois é por ele que passa toda a informação da simulação, embora também possa estar presente no *LocalCoordinator*.

3.5 Sincronização da memória partilhada

Para se desenvolver um sistema como o que é pretendido com este trabalho, com uma grande capacidade de intervenientes, surge a problemática da capacidade de um só computador não ser capaz de suportar um tão grande número de atores. Assim, existe a necessidade de se implementar o sistema como sendo um sistema distribuído, de forma a permitir ter um número de atores próximo do que será a realidade urbana. Foi então preciso analisar o sistema pretendido de forma a ficar escalável desde o início da sua concepção. Um dos primeiros problemas que surgiu foi como se iria sincronizar a memória de todas as máquinas presentes na distribuição de forma a mantê-las todas atualizadas, de modo a construir uma memória partilhada. A solução pensada foi assente na existência de uma rede LAN (*Local Area Network*) dedicada só a estas máquinas de forma a comunicarem entre si.

Ter uma memória partilhada significa que toda e qualquer alteração existente em qualquer uma das máquinas tem que ser difundida para todas as restante o quanto antes. Como é perceptível pela Figura 3.3, as alterações existentes numa simulação acontecerão sempre nos *LocalCoordinators*, uma vez que serão eles que irão possuir os atores que vão estar em constante movimento. Desta forma, os *LocalCoordinators* terão que estar constantemente a enviar informação sobre os seus nós para o *GlobalCoordinator*, por ser ele quem coordena toda a simulação, e para os restantes *LocalCoordinators*, pois os restantes atores existentes necessitam da informação dos atores com quem não partilham o *LocalCoordinator*. Para que a troca de informação seja feita de forma a garantir que em qualquer momento da simulação todos os componentes pertencentes ao seu núcleo estão atualizados, terá de haver um grande número de atualizações por minuto. Só assim será possível que atores presentes em diferentes *LocalCoordinators* saibam da existência e das movimentações de outros atores, necessário tanto para poderem comunicar como para ajustarem o seu movimentos.

De forma a que não haja atores que são mais vezes enviados que outros, considera-se uma atualização de cada vez que todos os atores associados aquele *LocalCoordinator* são enviados.

Perante o tipo de informação que necessitará de ser trocada e as necessidades de atualização, foi requerido que se encontrasse uma solução sólida e que satisfizesse todas as necessidades.

Dadas as características da transmissão em *multicast* IP, achou-se que usar este protocolo para esta situação seria muito vantajoso na medida em que as informações que cada *LocalCoordinator* tenha que enviar, não o fará apenas para o *GlobalCoordinator* mas sim diretamente para todas as entidades integrantes do sistema de uma só vez, bastando para isso que pertençam ao grupo *multicast* (Figura 3.5). Assim, além de se retirar carga ao *GlobalCoordinator*, diminuímos em muito a carga da rede, pois, no caso de toda a informação ser direcionada ao *GlobalCoordinator* e este a difundir para cada um dos *LocalCoordinators*, haveria uma grande redundância de informação a circular na rede.

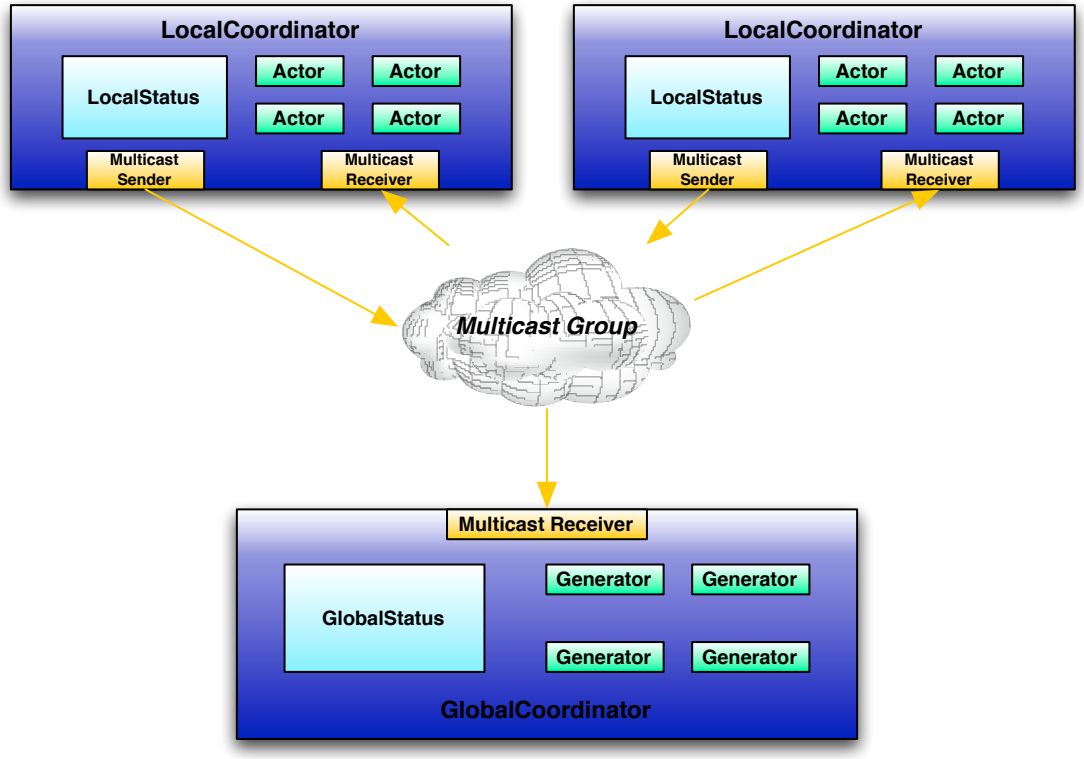

Figura 3.5 - Funcionamento do *multicast* IP no simulador

No entanto, o *multicast* IP tem por base o protocolo UDP, que não é seguro nem confirmado. Isto significa que na transferência das atualizações pode haver perdas e erros

que farão com que haja um extravio dos dados a chegar aos destinatários. Contudo, o número de atualizações por unidade de tempo que irão ser feitas pelos *LocalCoordinators* é muito grande, pelo que a perda pontual de uma quantidade pequena de informação não terá consequências catastróficas no desenrolar da simulação.

Contudo, como a rede de difusão será uma rede *ethernet*, significa que as suas tramas permitem um máximo de *payload* de 1500 bytes. Se a estes 1500 bytes disponíveis retirarmos os 28 bytes necessários para os cabeçalhos dos protocolos IP e UDP, restam 1472 bytes de informação útil. No entanto, este valor pode tornar-se pequeno com o decorrer da simulação para enviar toda a informação necessária. Todavia, esse problema é facilmente solucionável desde que se tenha fragmentação na origem. No entanto, com a fragmentação, no caso de perda de um dos pacotes toda a atualização é perdida, ou seja, uma grande quantidade de informação. Assim, optou-se por preencher os datagramas com o máximo de informação possível no próprio *LocalCoordinator*, sendo os datagramas posteriormente enviados individualmente. Desta forma, se se perder um dos pacotes enviados, não se perde a totalidade da atualização mas apenas um pequeno número de atores.

Analisando os prós e os contras do uso do *multicast*, verificou-se que o seu uso nos irá trazer um grande número de benefícios na difusão rápida e eficaz de toda a informação, perspectivando-se que seja o mais indicado para o que são as necessidades de sincronização de memória que este projeto exige.

3.6 Distribuição de carga

Uma vez que é ponto assente que o simulador terá de funcionar como um sistema distribuído em que haverá partilha de memória, surge a problemática de como será feita a distribuição da carga, ou seja, a distribuição dos atores pelos *LocalCoordinators*. Pela Figura 3.3, intuitivamente se compreende que essa distribuição será efetuada pelo coordenador da simulação, ou seja, pelo *GlobalCoordinator*.

Se no caso da atualizações de informações não será muito grave se se perder uma delas, o mesmo não se pode dizer da informação que é passada quando se quer criar um novo ator ou se pretende enviar um novo mapa necessário a esse mesmo ator. Para isso existirá a necessidade de se ter um protocolo de comunicações que seja fiável.

O protocolo TCP oferece a fiabilidade necessária para assegurar este tipo de troca de dados. Este protocolo não garante que a informação será entregue mas, em caso de falha, encarrega-se da retransmissão do que foi perdido, a menos que seja uma falha persistente e a conexão TCP seja quebrada.

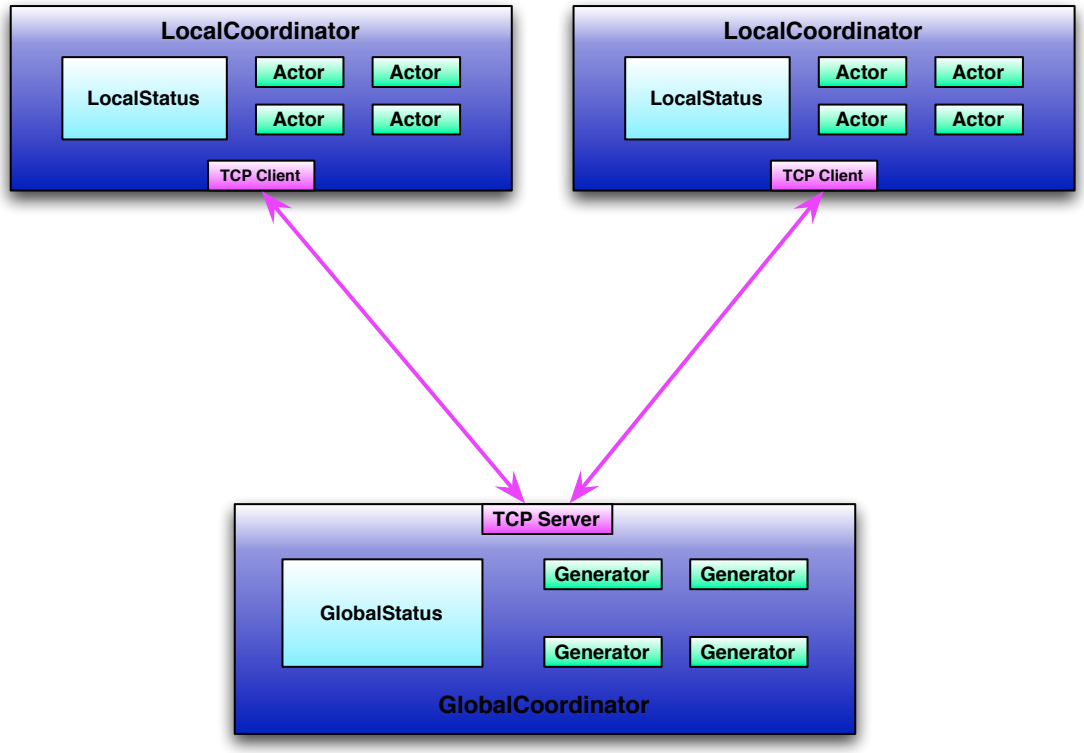

Figura 3.6 – Funcionamento do TCP no simulador

Assim, projetou-se que o protocolo TCP será usado para a distribuição da carga, ou seja, para enviar a ordem de criação dos novos *Actors* do *GlobalCoordinator* para o respectivo *LocalCoordinator*. A troca de mapas entre o *GlobalCoordinator* e o *LocalCoordinator*, integram o processo de criação de um ator, pelo que é contemplada neste protocolo.

Como pode existir a necessidade de haver trocas de informação fiáveis entre o *GlobalCoordinator* e a *Visualization*, este protocolo pode ser excecionalmente usado também para esses casos.

3.7 Equidade de carga

Sendo o sistema distribuído por diversos computadores, implica que seja feita uma gestão da carga de forma equitativa, caso contrário poderia haver computadores com grande quantidade de nós enquanto outros poderiam estar com uma quantidade muito menor.

Analisando quais as variáveis que se deverão ter em conta para que todas as máquinas estejam sempre ao mesmo nível, surgiram duas que poderão definir o escalonamento do sistema:

- Número de nós ativos no computador;
- Carga do processador do computador;

3.7.1 Número de nós ativos

O número de nós ativos foi pensado como uma metodologia de definição do escalonamento, pois se todos os computadores tiverem o mesmo número de nós ativos a carga será sempre equivalente em todos eles. Assim sendo, o *GlobalCoordinator* terá de saber quantos atores possui cada um dos *LocalCoordinators* e, em função disso, alocar os novos atores.

No entanto, esta distribuição pode não ser completamente justa se se tiver em conta que os diferentes nós serão de complexidade diferente. Por exemplo, um *tram* que apenas anda nos seus carris será menos complexo que um carro que pode andar por diversas estradas e que terá de saber todos os atributos de cada uma delas. Assim, poderá acontecer que um dos computadores fique mais carregado que outro por ter atores mais complexos.

Outro problema que esta distribuição poderá ter é o facto de os diferentes computadores terem características diferentes. Sendo feita a distribuição desta forma, qualquer que seja o *hardware* e o *software* dos computadores, todos ficarão com o mesmo número de nós. Contudo, podem existir computadores que não sejam capazes de suportar tantos nós como um outro que tenha componentes melhores, pelo que seria mais proveitoso que computadores com maior capacidade albergassem mais nós ou o computador que num determinado instante estivesse mais disponível, ou seja, com menos carga.

3.7.2 Carga do processador

Outro parâmetro analisado para ser feita a distribuição do sistema foi a carga atual do processador. O uso desta variável implica que todos os *LocalCoordinators* informem o *GlobalCoordinator* da carga do seu CPU para que, em função do seu valor, este atribua um novo ator ao computador mais livre. Para que esta informação esteja sempre atual, será enviado o valor da carga do CPU do *LocalCoordinator* para o *GlobalCoordinator*, a cada nova atualização dos atores.

Esta forma de distribuir a carga é mais eficiente que a anterior, pois a carga do CPU é influenciada pelos seu componentes. Assim, a distribuição seria feita de forma mais eficiente e cada computador seria aproveitado da melhor forma.

3.8 Mapas

Aquando do estudo para a implementação do BartUM Simulator, um dos parâmetros que ficou definido foi que as simulações a serem realizadas tinham de ser sobre mapas de cidades reais. Para isto ser possível, tinha de ser encontrado um formato de mapa que satisfizesse os requisitos.

Uma das exigências que existia sobre os mapas era serem simples e de fácil carregamento por parte do Java. Outra exigência seria o facto de esses mapas serem representados à custa de uma lista de pontos. Outro factor a ter em conta é o facto de ser possível haver pontos que não pertencessem às linhas, de forma a representarem pontos de interesse, como por exemplo lojas, monumentos, paragens, etc.

Após serem avaliados os parâmetros necessários a cumprir pelo formato dos mapas, foi analisada a possibilidade de se usar o formato *wkt*. Este formato, também usado pelo simulador *The ONE*, tem uma representação simples, apenas formado por pontos e linhas, e satisfaz as necessidades primordiais do simulador.

As linhas são constituídas por pontos separados por vírgulas, enquanto os pontos são constituídos pelas respectivas coordenadas *x* e *y* separados pelo caractere espaço. Para diferenciar as linhas dos pontos é usada uma palavra no início de cada linha do ficheiro, como se pode ver pelo exemplo de duas possíveis linhas do um ficheiro, apresentado de seguida:

```
LINESTRING (5.644471698113208 -72.4011132075471, 14.668018867924529 -
54.35401886792445, 65.42547169811321 -63.37756603773577)
POINT (2552447.349579492 6673343.299785728)
```

Uma outra característica muito vantajosa que este tipo de mapas permite é o facto de se poder inserir pontos numa linha, fazendo com que se possa atribuir o mesmo ponto a duas linhas de forma ser criado um ponto interceptante entre elas.

Para se formarem e alterarem os mapas tem de ser usada uma aplicação, denominada de *OpenJUMP* que pode ser descarregada em (OpenJUMP GIS) e que é multiplataforma. Uma outra característica que levou a optar por esta solução foi o facto de esta aplicação

permitir que seja aberta uma imagem para ficar em *background* enquanto se desenham as linhas e pontos dos mapas. Este facto permite que o mapa por nós desenhado fique bastante próximo da realidade.

3.9 Funcionamento do sistema

Uma vez alcançada uma arquitetura estável, será necessário estabelecer como será o funcionamento do sistema. Descrever como irá funcionar o sistema será um auxílio importante para a sua implementação uma vez que se tem definido o objectivo a que se pretende chegar.

Como já foi dito anteriormente, na secção 3.4, o simulador terá 8 entidades principais que, embora funcionem de forma independente, trabalham para alcançarem um objectivo comum. Tal como foi definido aquando da definição da arquitetura do sistema, o *GlobalCoordinator* e os vários *LocalCoordinators* vão ficar alojados em diferentes máquinas. Ter estas duas entidades em computadores diferentes significará que funcionarão como aplicações diferentes, logo, terão diferentes formas de inicialização.

No *software* existirá uma classe responsável pelo arranque do sistema que terá de seguir uma ordem de procedimentos a ser definida. Isto é um factor importante para que o sistema tenha um suporte inicial seguro e íntegro. Assim, na definição da inicialização do sistema tem de se ter em conta situações que não poderão ocorrer, como é o caso de não se poder criar um novo *Actor* sem que tenha sido instanciada a lista onde este será guardado, ou um *Actor* necessitar de se mover sem que o *LocalStatus* possua o respectivo mapa ou mapas.

Desta forma, foi feito um levantamento das possíveis dependências que cada uma das entidades, *GlobalCoordinator* e *LocalCoordinator*, poderão ter de forma a ser encontrada uma possível sequência de inicialização para cada uma delas.

3.9.1 GlobalCoordinator

De seguida será exposta a sequência de inicialização do *GlobalCoordinator*, assim como uma descrição do seu funcionamento ao longo de uma simulação.

3.9.1.1 Inicialização

Tendo em consideração aquilo que será o funcionamento do *GlobalCoordinator* foram encontradas as seguintes dependências:

1. Não poderão ser criados/carregados mapas sem que se tenha sido criado um local onde os guardar, neste caso a entidade *GlobalStatus*;
2. Nunca poderão ser criados *Actors* sem que os mapas já existam no *GlobalStatus*;
3. Antes de se criar/iniciar qualquer uma das interfaces de rede tem que ser iniciado o *logging* da rede;
4. Sem que seja aceite uma conexão de um *LocalCoordinator* não tem sentido que sejam criados e iniciados os *Generators* pois não terá *LocalCoordinator* onde associar os atores gerados;

Partindo das premissas anteriores, foi definida uma ordem de arranque para todo o sistema presente no computador que alojará o *GlobalCoordinator* (Figura 3.7).

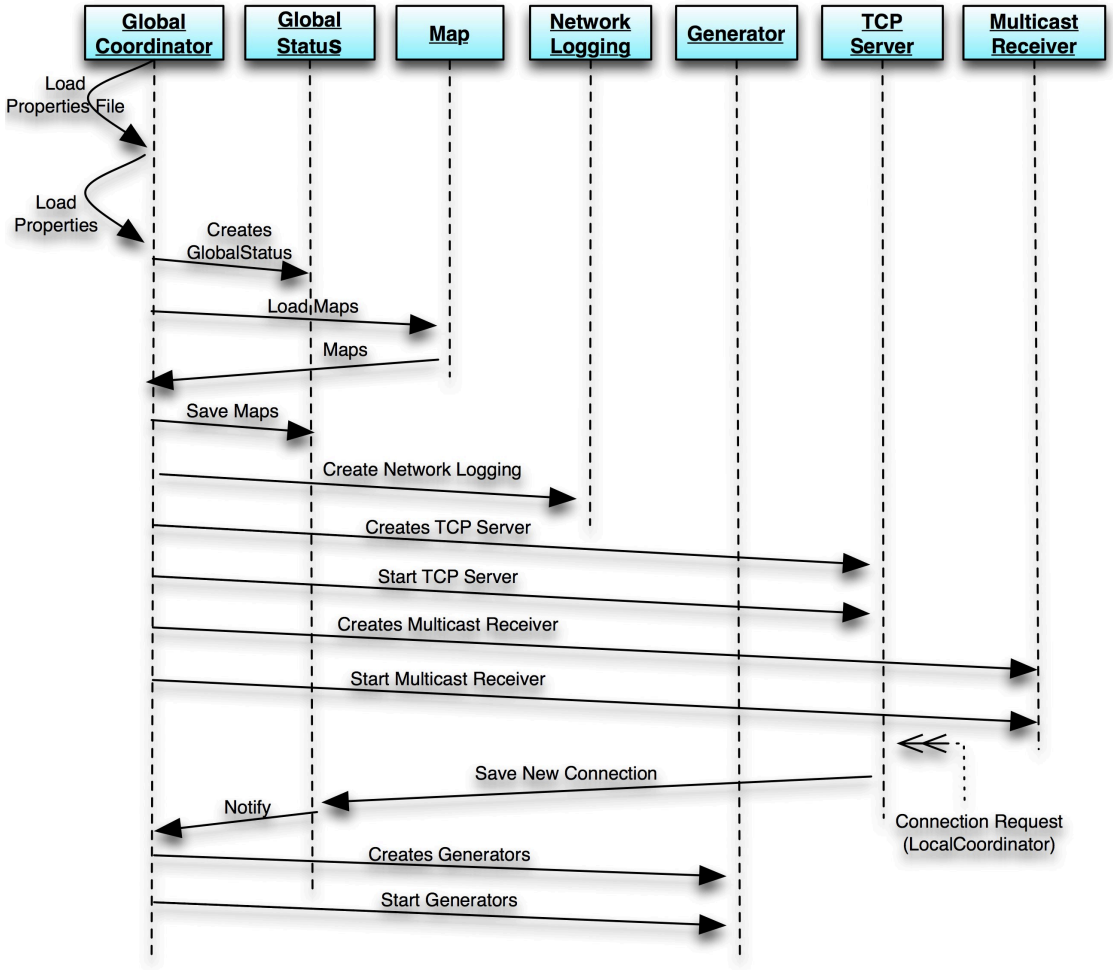

Figura 3.7 - Diagrama de sequência do arranque da entidade *GlobalCoordinator*

Para a inicialização desta parte do sistema existirá uma classe *GlobalCoordinator* que será responsável pelo arranque do mesmo. A primeira tarefa dessa classe será carregar o ficheiro, do tipo *properties*, com as configurações iniciais do sistema. Uma vez carregado o ficheiro, a classe *GlobalCoordinator* fará o carregamento das propriedades que constarão no ficheiro para serem utilizadas.

Após estarem carregadas as variáveis necessárias ao arranque do sistema, será inicializado o *GlobalStatus*. Inicializado o *GlobalStatus* já será possível ao *GlobalCoordinator* carregar e guardar os mapas necessários à simulação, cumprindo assim a primeira premissa.

Uma vez carregados e guardados os mapas no *GlobalStatus*, será o momento de se começar a ativar as interfaces de rede necessárias ao funcionamento do sistema. A primeira coisa a ser inicializada será a classe que fará o *logging* da rede, a classe *Network Logging*, sendo esta classe a responsável pelo *Reporting* da rede. Depois de iniciado o *logging* das

interfaces de rede poder-se-á começar a estabelecer conexões entre o *GlobalCoordinator* e os *LocalCoordinators*, assegurando assim a dependência número 3. Das duas interfaces de comunicação necessárias ao *GlobalCoordinator*, *TCP Server* e *Multicast Receiver*, a primeira que será efectuada será o *TCP Server*, pois é a partir deste que se iniciarão as conexões com os *LocalCoordinators*, seguida da inicialização e arranque da interface *Multicast Receiver*.

Estando já preparado para receber conexões TCP da parte dos *LocalCoordinators*, a classe *GlobalCoordinator* ficará à espera que alguma conexão seja efectuada com sucesso para terminar o arranque do sistema. Logo que seja conseguida uma conexão, via TCP, com um *LocalCoordinator*, ao *GlobalCoordinator* será informada de que poderá finalizar a inicialização. Assim sendo, através das informações constantes no ficheiro *properties*, são iniciados e arrancados os *Generators* e dada como finalizada a inicialização. Uma vez que por cada *Generator* que é criado existe a possibilidade de serem criados novos mapas resultantes de junções de mapas já existentes, a junção desses mapas é feita antes do arranque de cada *Generator*, fazendo com que as dependências 2 e 4 sejam cumpridas.

3.9.1.2 Funcionamento Geral

A partir do momento em que a inicialização se encontra concluída, o sistema entrará numa fase estacionária em que terá procedimentos feitos de forma cíclica.

Assim, a interface *TCP Server* estará constantemente à espera de novas conexões, sejam elas vindas de *LocalCoordinators* ou de *Visualizations*. Sempre que chegue um novo pedido de conexão, o *TCP Server* notificará o *GlobalCoordinator*, informando-o de que tipo é a nova conexão para que saiba como a usar e se assim o entender, criará um novo servidor *slave* para responder aos pedidos que forem feitos pelo cliente. Esse *slave server* ficará ativo a responder ao pedidos até que o cliente feche a conexão ou esta seja quebrada.

Por outro lado, o *Multicast Receiver* estará constantemente à espera de novas actualizações, sendo que de cada vez que receber uma nova mensagem com nova informação terá que a disponibilizar ao *GlobalStatus* para que este se mantenha actual. Esta interface terá de estar constantemente ativa e à escuta de novas informações.

Os *Generators*, depois de serem ativados e de ser feito o *merge* dos mapas que os seus *Actors* irão necessitar, estarão constantemente a gerar um número aleatório entre 0 e 1. No caso de o número que foi gerado estar dentro de um intervalo, definido inicialmente,

será enviada uma notificação ao *GlobalCoordinator* com a informação de que tipo de ator terá que enviar para um dos *LocalCoordinators* existentes. De forma a que a cadência com que os atores são gerados seja o mais aproximado possível da realidade, o intervalo a que o número aleatório terá que pertencer será diferente em função do tipo de *Actors* que o *Generator* estiver a gerar e também da localização inicial dos mesmos. Além de intervalos diferentes, outro factor que irá auxiliar o realismo da cadência da geração de *Actors* será o ritmo com o *Generator* gerará o valor aleatório, sendo diferente para cada tipo de *Generator*.

O *GlobalCoordinator* será quem, no normal funcionamento, terá mais tarefas associadas a si, já que ele terá de controlar toda a simulação. Ele terá de ter uma tabela em que guardará o IP de cada um dos *LocalCoordinators* que já se ligaram a ele juntamente com a informação referente à sua carga, para que possa distribuir os *Actors* de forma eficiente. Além disso, terá que estar a guardar constantemente as actualizações dos *Actors* que lhe vão chegando juntamente com as mudanças do valor da carga. Por fim, o *GlobalCoordinator* terá de albergar todos os mapas que forem necessários à simulação e disponibiliza-los sempre que solicitado pelos *LocalCoordinators* ou pela *Visualization*.

3.9.2 LocalCoordinator

Tal como foi elaborado para o *GlobalCoordinator*, também será exposta para o *LocalCoordinator* a sua sequência de inicialização assim como a descrição do seu funcionamento ao longo de uma simulação.

3.9.2.1 Inicialização

Tal como foi feito para o *GlobalCoordinator*, também para o *LocalCoordinator* foi feito um levantamento das dependências, considerando aquilo que será o seu funcionamento, tendo sido encontradas as seguintes dependências:

1. Não poderão ser criados/iniciados *Actors* sem que se tenha sido criado um local onde os guardar, neste caso a entidade *LocalStatus*;
2. Antes de se criar/iniciar qualquer uma das interfaces de rede tem que ser iniciado o *logging* da rede;
3. Sem que seja criado pelo menos uma *Actor* não tem sentido que seja criada e iniciada a interface *Multicast Sender*, uma vez que não existe informação para enviar;

Partindo dos princípios anteriores foi definida a ordem de arranque para todo o sistema presente na máquina que suportará o *LocalCoordinator* (Figura 3.8).

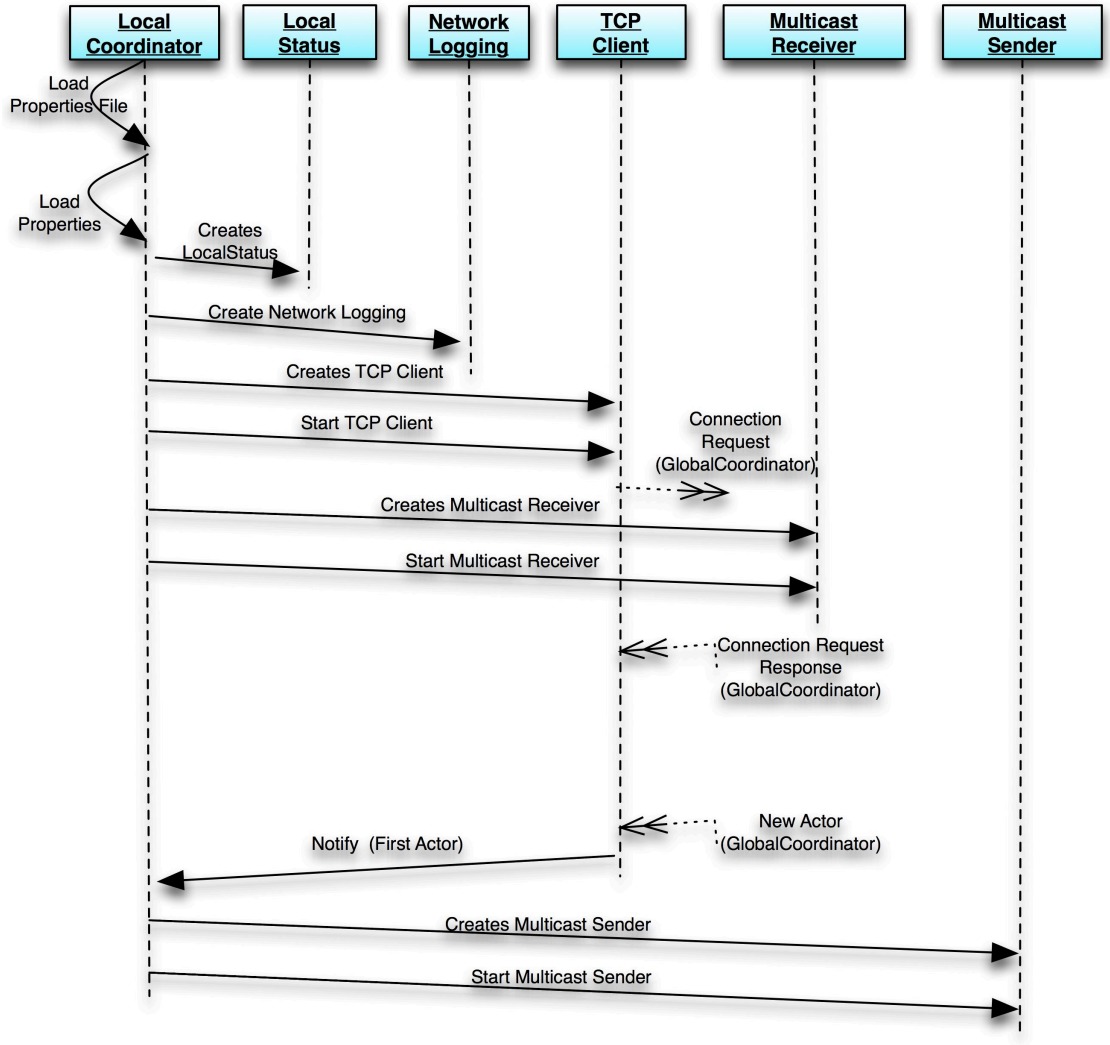

Figura 3.8 - Diagrama de sequência do arranque da entidade *LocalCoordinator*

Para a inicialização do sistema existirá uma classe *LocalCoordinator* que será responsável pelo arranque do mesmo, tal como acontecerá com o *GlobalCoordinator*. A primeira tarefa dessa classe será carregar o ficheiro, do tipo *properties*, com as configurações iniciais do sistema. Uma vez carregado o ficheiro, a classe *LocalCoordinator* fará o carregamento das propriedades que constarão no ficheiro para serem utilizadas.

Após estarem carregadas as variáveis necessárias ao arranque do sistema, será inicializado o *LocalStatus.* Inicializado o *LocalStatus*, já será possível receber ordens de criação de *Actors* pois já existe onde os guardar, garantindo o comprimento da dependência número 1.

Uma vez criado o *LocalStatus*, será o momento de se começar a ativar as interfaces de rede necessárias ao funcionamento do sistema. A primeira classe a ser inicializada será, da mesma forma que acontece no *GlobalCoordinator*, a que fará o *logging* da rede, a classe *Network Logging*. Depois de iniciado o *logging* das interfaces de rede poder-se-á começar a estabelecer conexões entre o *LocalCoordinator* e o *GlobalCoordinator*, assegurando assim a dependência número 2.

Das três interfaces de comunicação necessárias ao *LocalCoordinator*, *TCP Client, Multicast Sender* e *Multicast Receiver*, a primeira que será efectuada será o *TCP Client*, pois é a partir deste que se iniciará a conexão com o *GlobalCoordinator*, seguida da inicialização e arranque da interface *Multicast Receiver*, como se pode ver na Figura 3.8.

Logo que a interface *TCP Client* é iniciada fará um pedido de conexão ao *GlobalCoordinator* existente na rede de forma a que este saiba da sua existência e o coloque na lista de candidatos a receber novos *Actors*. Uma vez feito o pedido de ligação com o *GlobalCoordinator*, a classe *LocalCoordinator* ficará a aguardar pela notificação para o nascimento do primeiro *Actor*. Aquando deste acontecimento, o *LocalCoordinator* criará e ativará a interface *Multicast Sender* e dará por finalizado a arranque desta parte do sistema e garantindo as dependência descritas.

A ativação e arranque da interface *Multicast Receiver* será feita logo de seguida a ser arrancada a interface *TCP Client*, ou seja, não terá de esperar que seja estabelecida qualquer conexão para o seu arranque. Isto acontecerá pois desta forma o *LocalCoordinator* poderá receber atualizações que estejam a ser enviadas para o grupo *multicast* fazendo com que quando receber ordem para criar o primeiro *Actor* já terá algumas informações sobre os atores existentes na simulação.

3.9.2.2 Funcionamento Geral

Uma vez iniciado o *LocalCoordinator*, o sistema entrará numa fase estável e em que o funcionamento se tornará de certa forma cíclico.

À medida que vão chegando novos *Actors* ao *LocalCoordinator* eles serão inicializados. O seu funcionamento será idêntico seja qual for o tipo de ator e estará constantemente em movimento. Esse movimento terá que ser feito sobre os mapas que lhe correspondem e terá de ter em conta várias variáveis como é o caso do seu tipo, da sua velocidade, da sua

localização e os *Actors* à sua volta. Sempre que um ator terminar uma movimentação ele terá de reportar as alterações ao *LocalStatus*.

O *TCP Client* terá que se manter ativo durante toda a simulação pois é por esta interface que o *LocalCoordinator* será notificado de novos *Actors* que tenha de fazer nascer. Será também através desta interface que o *GlobalCoordinator* e o *LocalCoordinator* trocarão informações sobre os mapas e será também por ela que serão enviados os mapas. Em caso de algum tipo de falha terá que ser instanciada uma nova interface *TCP Client* de forma a que o *LocalStatus* continue a receber novos *Actors*.

Além do *TCP Client*, o *LocalCoordinator* terá ainda mais duas interfaces de comunicação, e serão elas o *Multicast Sender* e o *Multicast Receiver*. No caso do *Multicast Sender* será uma interface que a cada intervalo de tempo, definido no início da simulação, enviará para o grupo *multicast* a informação do estado atual dos *Actors* associados a si. Estas atualizações irão impedir que haja colisões entre *Actors* que estejam em *LocalCoordinators* diferentes.

Já o *Multicast Receiver* será a interface que estará à escuta no grupo *multicast* de forma a receber as informações que serão enviada pelos outros *LocalCoordinators*. Assim, sempre que uma nova mensagem de informação chega a esta interface, ela notificará o *LocalStatus* e disponibilizará a informação acabada de chegar para que possa ser guardada.

Por fim, o *LocalCoordinator* será a base local do simulador e terá um funcionamento semelhante ao do *GlobalCoordinator*. Tal como acontece no *GlobalStatus*, o *LocalStatus* terá de ser capaz de guardar a informação dos *Actors* que não estão associados a si e que chegarão dos outros *LocalCoordinators* e também ser capaz de guardar os mapas vindos do *GlobalCoordinator*, sendo que neste caso apenas terá de guardar os mapas que são necessários aos seus atores e não todos eles. Esta entidade tem que ser capaz de, através da informação base dos *Actors* que lhe é enviada pelo *GlobalCoordinator*, criar novos *Actors* de todos os tipos existentes. Finalmente, o *LocalStatus* terá que ser capaz de guardar as actualizações dos *Actors* associados a si assim como ser capaz de indicar aos seus *Actors* quais os seus vizinhos, sempre que estes o solicitarem, de forma a evitar colisões.

4 Implementação

Finalizada a análise do sistema a desenvolver, será neste capítulo analisada a implementação do que foi planeado no capítulo 3 (Bartolomeu_Urban Mobility Simulator (BartUM)). A implementação do simulador foi realizada na linguagem de programação Java.

De forma a que a implementação do simulador ficasse o mais possível organizada e facilmente perceptível onde se encontrará cada classe, o *software* foi dividido em 5 *packages*, todos eles com o prefixo *um.simulator.*: *core*, *map*, *communications*, *actor* e *visualization*. Cada um deste *packages* terá as seguintes classes:

- *um.simulator.core.*:
    - *GlobalStatus*;
    - *GlobalCoordinator*;
    - *LocalStatus*;
    - *LocalCoordinator*;
    - *Generator*;
    - *NetworkLogging*;
- *um.simulator.map.*:
    - *Global_Map*;
    - *Map_Line*;
    - *Map_Point*;
    - *Point_Line*;
- *um.simulator.communications.*:
    - *MulticastSender*;
    - *MulticastReceiver*;
    - *TCPClient*;
    - *TCPServer*;
    - *TCPServerPersonal*;
- *um.simulator.actor.*:

- *ActorStatus*;
- *Actor*;
- *ActorTram*;
- *ActorCar*;
- *ActorPedestrian*;
- *ActorTrain*;
- *ActorCycle*;
- *ActorBus*;

- *um.simulator.visualization.*:
  - *Visualization*.

Para ser mais perceptível a utilidade de cada uma das classes, assim como conhecer quais os seus métodos e variáveis, é apresentado nas Figura 4.1 e Figura 4.2 os diagramas de classes do BartUM Simulator.

Como já foi dito anteriormente, este trabalho foi partilhado por 3 pessoas, pelo que neste capítulo só será especificada a implementação referente aos objetivos definidos nesta dissertação. Assim, será descrita, de seguida, a implementação dos *packages core*, *map*, *communications* e uma classe do *actor*.

4.1 Diagramas de classes

Uma vez que já ficou finalizada a análise do simulador a implementar, teve que ser feita uma projeção do que iria ser cada uma das classe a desenvolver. De forma a realizar uma implementação estável e sem redundâncias, com o auxílio das várias interações já identificadas no capítulo anterior, foram desenvolvidos dois diagramas de classes, um para o *GlobalCoordinator* (Figura 4.1) e outro para o *LocalCoordinator* (Figura 4.2). Esses dois diagramas foram um importante auxílio no desenvolvimento de cada uma das classes presentes no simulador.

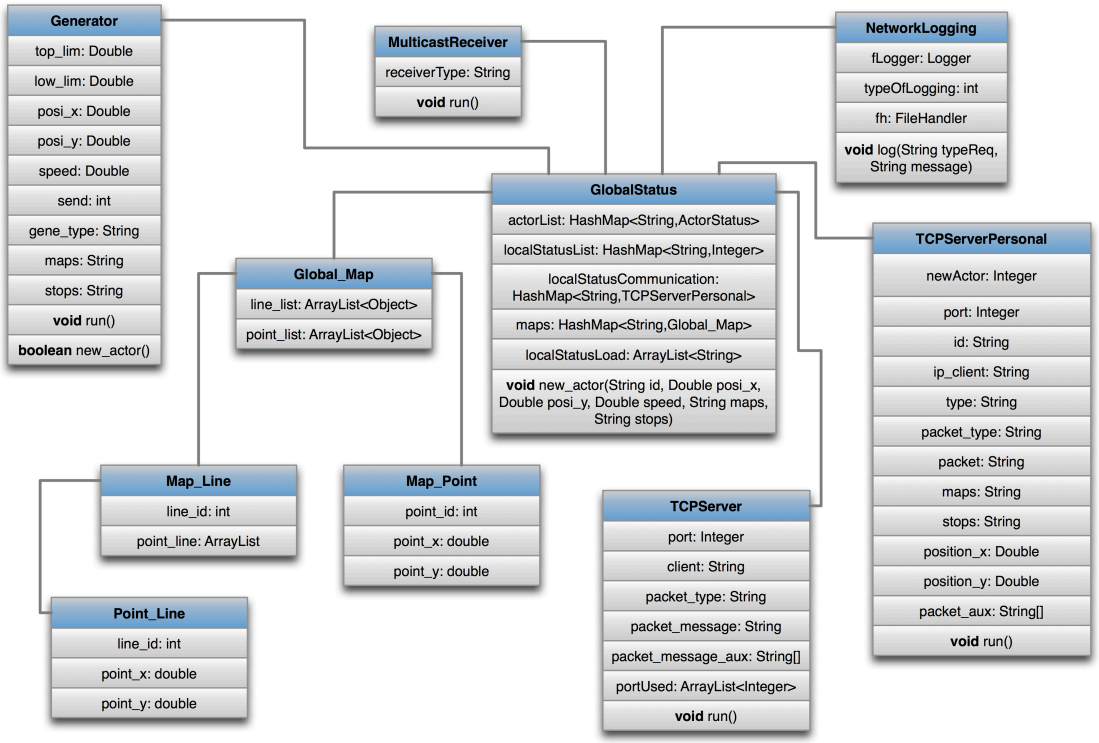

Figura 4.1 - Diagrama de classes do *GlobalCoordinator*

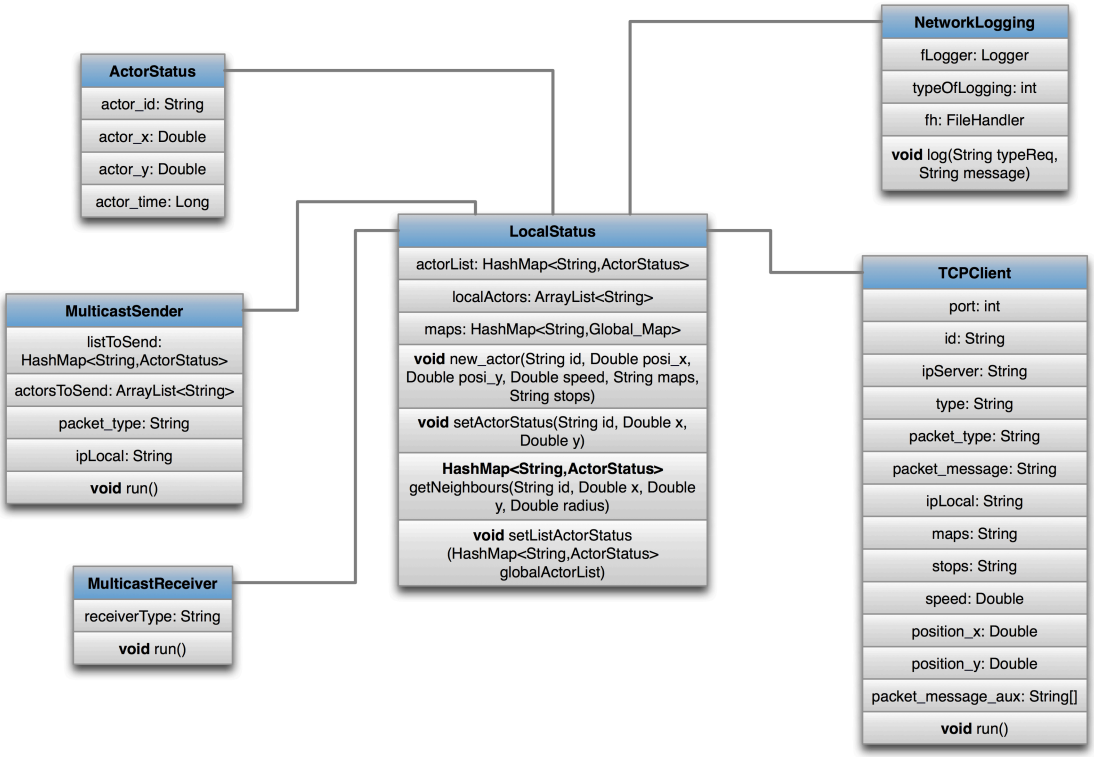

Figura 4.2 - Diagrama de classes do *LocalCoordinator*

4.2 Core

O *package um.simulator.core*, de agora em diante denominado apenas de *core*, é aquele que é composto por todas as classes que pertencem ao núcleo do funcionamento do sistema.

4.2.1 GlobalStatus

Anteriormente foi explicado o funcionamento da classe *GlobalStatus* e, nessa descrição, foi dito que a sua função é guardar toda a informação existente da simulação a decorrer.

4.2.1.1 Variáveis globais

Como se pode ver na Figura 4.1, esta classe possui 5 listas onde mantém guardada toda a informação da simulação que esta a decorrer. Essas foram as listas que se consideraram estritamente necessárias para que fosse usado o mínimo de estruturas possíveis. Assim, o *GlobalStatus* tem a `actorList` que guarda toda a informação essencial de cada um dos *Actors* ativos na simulação, a `localStatusList` que guarda a informação necessária para a comunicação com todos os *LocalCoordinators* em funcionamento no sistema, a `localStatusCommunication` que mantém as interfaces de comunicação com os *LocalCoordinators* acessíveis a ser usadas bastando para isso saber o respectivo IP, a `maps` que mantém todos os mapas carregados e necessários para o decorrer da simulação e, por fim, a `localStatusLoad` que contém a lista dos *LocalCoordinators* existentes juntamente com a carga atual do seu sistema. Todas estas listas permitem que outros objetos possam aceder a sua informação diretamente, ou seja, sem ser através da criação de um objeto *GlobalStatus*.

A `actorList` foi implementada como sendo uma espécie de tabela, em que a primeira coluna é o *id* de cada *Actor* e a segunda é um objeto, do tipo `ActorStatus`, que será descrito mais adiante (secção 4.5.1), que guarda toda a informação necessária de um *Actor*. De cada vez que um *Actor* deixar de estar ativo tem que ser retirado desta lista para que estejam nesta lista apenas os nós ativos. Como cada nó tem um *id* único, nunca haverá entradas repetidas na tabela.

A `localStatusList` foi também implementada como sendo uma tabela e em que a sua chave é o IP de cada *LocalCoordinator*, associado à porta pela qual a comunicação TCP esta a ser feita. Uma vez que um IP só pode existir uma vez em cada LAN, a unicidade da

chave fica garantida, e ao ter associada a porta pela qual se esta a fazer a comunicação permite que não seja atribuída uma porta igual a outro IP, pois há o conhecimento de todas as portas que estão abertas.

A `localStatusCommunication`, também ela uma tabela, foi criada para guardar os objetos usados para a comunicação TCP. A chave desta tabela é o IP do *LocalCoordinator* que fica assim associada ao objeto `TCPServerPersonal`, descrito mais adiante (secção 4.4.2.4), usado para a comunicação TCP com o referido *LocalCoordinator*. Isto faz com que sempre que se quiser comunicar com um determinado *LocalCoordinator* apenas se precise de saber o seu IP, obtendo como resposta o objeto a usar para comunicar.

A `localStatusLoad`, ao contrário das restantes, é apenas uma lista composta pelos IP dos *LocalCoordinators* juntamente com a última atualização da sua carga enviada ao *GlobalCoordinator*, ou seja, cada entrada desta lista é o IP mais a carga, separados pelo caractere '/', de todos os *LocalCoordinators* que estão conectados com o *GlobalCoordinator*.

A `maps` é uma tabela que contém todos os mapas usados para a simulação que está a decorrer. Para isso, esta tabela tem como chave o nome padrão de *Map.x*, em que o *x* é o número decimal da ordem em que foi carregado, ou seja, começa em 1 e vai sendo incrementado a cada novo mapa que é carregado. Associada a essa chave está um objeto `Global_Map`, explicado posteriormente (secção 4.3.1), que contém as informações relativas a esse determinado mapa.

4.2.1.2 Funções

O objeto *GlobalStatus*, além das funções habituais de *get* e *set*, apenas possui um método que é evocada por outros objetos, quando necessitam. Este método é responsável por decidir a qual *LocalCoordinator* será enviado um novo *Actor*. Este método foi implementado no *GlobalStatus*, pois é lá que esta guardada a informação da carga dos *LocalCoordinator* ficando assim facilitado a acesso a essa mesma informação. Só com a ação do *GlobalStatus* na divisão dos atores é possível que seja feita uma distribuição equilibrada.

new_actor

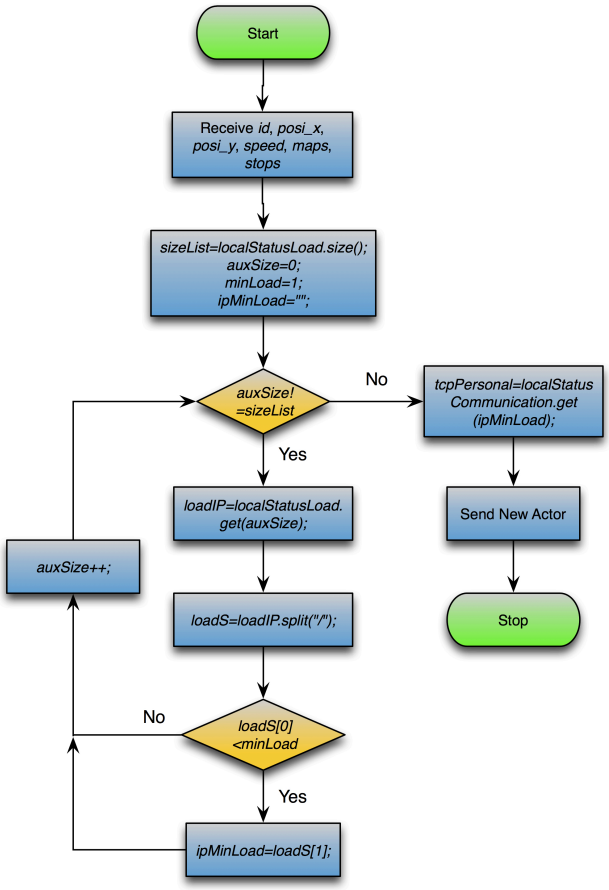

Figura 4.3 - Fluxograma da função *new_actor*

Como se pode verificar no Figura 4.3, esta função tem de percorrer toda a lista `localStatusLoad`, separando o IP da carga, de forma a que consiga encontrar o *LocalCoordinator* que tem uma carga menor. Quando o *LocalCoordinator* com menor carga for encontrado, é usado o seu IP para se obter a interface `TCPServerPersonal` presente na tabela `localStatusCommunication` de forma a que lhe seja enviado o novo *Actor*. Esta função é evocada de cada vez que um dos *Generators* dá ordem de nascimento para um novo nó. Para que o *GlobalStatus* saiba todas as informações que tem de passar ao *LocalCoordinator*, os *Generators* passam-na como parâmetro no evocar da função. Como auxílio na compreensão do fluxograma apresentado na Figura 4.3, é demonstrado um exemplo prático de uma evocação da função na Figura 4.4.

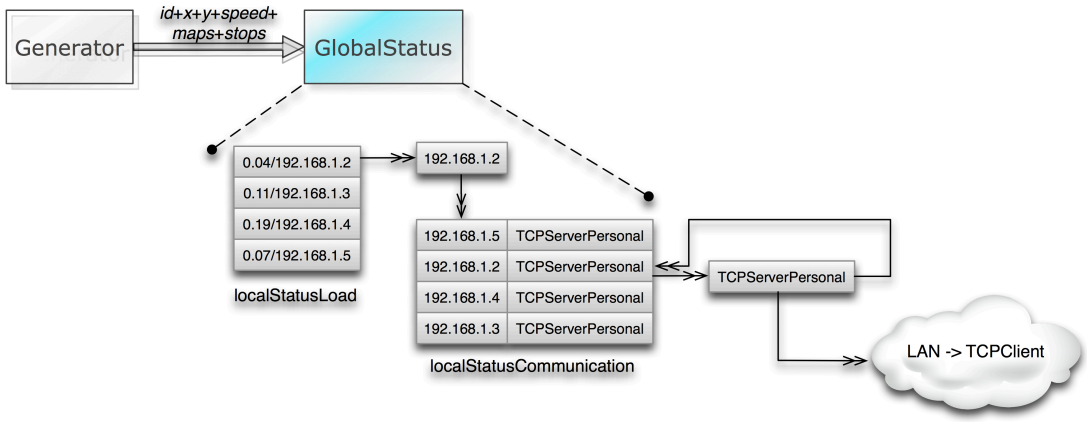

Figura 4.4 - Exemplo de funcionamento da função *new_actor*

No entanto, esta função pode levar a que surjam alguns conflitos no seu acesso. Um exemplo disso será um *Generator* dar ordem para criar um *Actor* quando um outro já deu essa mesma ordem. O que pode acontecer é que o *GlobalStatus* poderia atender o novo pedido sem que tivesse acabado o que já estava a ser atendido, fazendo com que o novo *Actor* não seja criado ou seja criado de forma incorreta, como por exemplo, haver confusão de variáveis e nascer um *Actor* do tipo *Ped* com os mapas de um *Tra*. Para ultrapassar este inconveniente foi usada uma funcionalidade do Java (`synchronized`) que permite que apenas um processo de cada vez aceda a essa função, ou seja, a função só atende outro processo quando terminar o que já estava a atender.

4.2.2 GlobalCoordinator

A classe *GlobalCoordinator* é aquela que inicializa todo o sistema no computador onde está alojado o *GlobalStatus*. A sua ordem de inicialização já foi abordada anteriormente (secção 3.9.1.1), pelo que a implementação desta classe foi apenas feita seguindo o que foi projetado.

Tal como foi dito aquando da explicação do arranque do sistema, esta classe utiliza um ficheiro para carregar informações necessárias para o arranque do sistema. Esse ficheiro tem propriedades muito específicas e pode ser usado de forma a se conseguir aceder rapidamente à informação nele contida.

4.2.2.1 Ficheiro Properties

O ficheiro que tem as informações necessárias ao arranque é do tipo *properties*, que permite que sejam aproveitadas algumas funcionalidades presentes no Java para este tipo de ficheiros.

O ficheiro é carregado para uma instância de um classe do tipo `Properties` fazendo com que depois se possa aceder de forma simples e direta à informação lá contida.

Para que a informação seja então de fácil acesso, o ficheiro *properties* tem que ser escrito de uma forma padrão, ou seja, com uma espécie de *tags* que são depois evocadas para que seja devolvido o seu valor, como se pode ver no exemplo a seguir.

```
#Generators
Generator.Number=6
Generator.1=Tra
Generator.2=Ped
Generator.3=Car
Generator.4=Cyc
Generator.5=Bus
Generator.6=Tri
```

No exemplo apresentado, as *tags* utilizadas são `Generator.`*x* e o valor associado a cada *tag* é o que está depois do símbolo '='. Neste padrão textual, todas as linhas que sejam iniciadas com o caractere '#' são considerados comentários pelo que são ignoradas quando o ficheiro é carregado.

```
n_generator=Integer.parseInt(prop.getProperty("Generator.Number"));
        generators.add(prop.getProperty("Generator."+t));
```

As duas linhas de código apresentadas mostram como é possível aceder-se à informação que está presente no ficheiro de uma forma simples, usando apenas a *tag* com que foi escrito no ficheiro. A informação que é retirada do ficheiro encontra-se toda no formato de `String`, sendo que no caso de se querer que essa informação seja de outro tipo se tenha que fazer o respectivo *parse*.

4.2.3 LocalStatus

Uma vez que o *LocalStatus* foi criado a partir do *GlobalStatus* estes têm um funcionamento semelhante. No entanto, no caso do *LocalStatus* existe uma relação direta com os nós presente na simulação, o que obriga a ter um número maior de funções.

4.2.3.1 Variáveis globais

Nesta classe não foram usadas tantas listas/tabelas como no *GlobalStatus* uma vez que não é função do *LocalStatus* guardar toda a informação da simulação, mas sim aquela que lhe é diretamente necessária. Contudo, as duas tabelas e a listas que foram implementadas, tal como no *GlobalStatus*, permitem que outros objetos possam aceder à sua informação diretamente. Na Figura 4.2 são apresentadas todas as variáveis globais necessárias na sua implementação.

A tabela `actorList` é usada para o *LocalStatus* guardar as informações necessárias de todos os *Actors* ativos na simulação, incluindo os seus. Esta tabela foi implementada de forma semelhante que a tabela `actorList` do *GlobalStatus*, tendo como chave o *id* do *Actor*, associada ao `ActorStatus` correspondente. De cada vez que o *LocalStatus* recebe uma atualização, todos os *Actors* recebidos são colocados nesta lista, fazendo assim com que a informação relativa aos atores esteja toda concentrada no mesmo local.

Como quando é enviada uma atualização são enviados apenas os atores que estão associados a esse *LocalCoordinator*, surge o problema de como é que o *LocalStatus* seleciona apenas os nós que estão associados a si de toda a tabela `actorList`. Para que isso seja possível, cada *LocalStatus* tem de ter uma `localActors`, que consiste numa lista que contém todos os *id* dos atores associados a si. Assim, de cada vez que uma atualização do *LocalCoordinator* é enviada, apenas vai enviar os seus próprios *Actors*.

Tal como o *GlobalStatus*, também o *LocalStatus* precisa de uma tabela para guardar os seus mapas, a tabela `maps`. Contudo, o *LocalStatus* não precisa de albergar todos os mapas da simulação, apenas necessita daqueles que os seus atores vão precisar para se mover. Assim, um dos campos de informação que o *LocalStatus* recebe para a criação de um novo *Actor* é o *id* dos mapas que ele precisa de possuir para que aquele novo nó se possa movimentar. No entanto, com o desenrolar da simulação, como não há seletividade por tipo de ator, os *LocalCoordinator* podem acabar por possuir todos os mapas.

4.2.3.2 Funções

As principais diferenças entre os *LocalStatus* e o *GlobalStatus* estão nos métudos que cada um possui. Enquanto no *GlobalStatus* apenas foi implementada uma função, no *LocalStatus* houve a necessidade de serem implementadas quatro. Isto acontece devido ao facto de haver uma interação direta entre os atores da simulação e o *LocalStatus*. Se se pode

considerar que o *GlobalStatus* é o apoio direto dos *LocalStatus*, os *LocalStatus* são o apoio direto ao *Actor*, pelo que têm de possuir as funções necessárias às suas necessidades.

new_actor

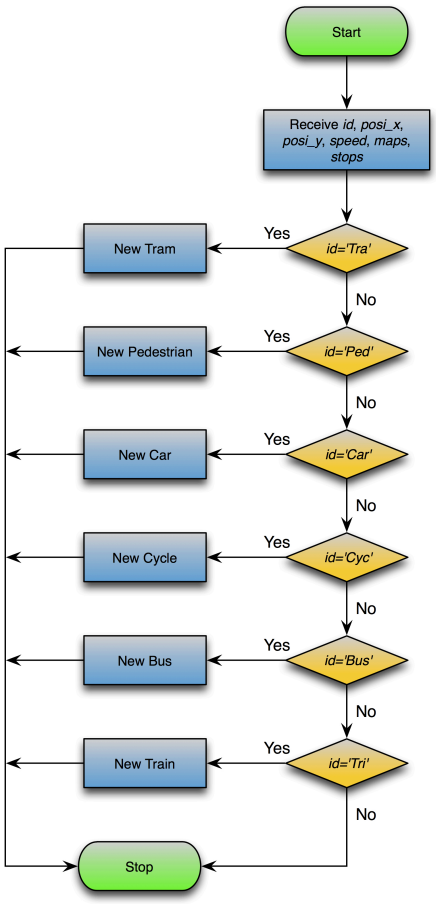

Figura 4.5 - Fluxograma do funcionamento da função *new_actor*

A função `new_actor` foi implementada para criar novos atores à medida que vai recebendo ordens para isso. Em função do tipo de *Actor* que tem que ser criado é iniciado esse tipo específico de *Actor*. Como se pode ver na Figura 4.5, para que um novo ator seja criado o *LocalStatus* recebe, nesta função, um série de parâmetros:

- `String id` – é o que define o tipo de *Actor* que é criado. Além dos 3 primeiros caracteres, que definem o tipo de nó, tem sempre associado um valor numérico, com pelo menos 2 dígitos, onde o primeiro é o identificador do *Generator* e os restantes para diferenciar os *Actors*. Exemplo: Tra17 – é o *tram* número 7 gerado pelo *Generator* de *tram* 1;

- `double posi_x` – é a coordenada *x* em que o *Actor* tem que iniciar o seu movimento. Esta coordenada é igual para todos os atores gerados pelo mesmo *Generator*;
- `double posi_y` – é a coordenada *y* em que o *Actor* tem que iniciar o seu movimento. Esta coordenada, tal como acontece com o *x*, é igual para todos os atores gerados pelo mesmo *Generator*;
- `double speed` – é o valor da velocidade inicial do *Actor* a ser criado. No entanto, com o desenrolar da simulação este valor vai sendo alterado, pois pode acontecer de ter de parar num cruzamento, ou mudar as condições da estrada em que circula, etc.;
- `String maps` – é o nome dos mapas que o *LocalStatus* precisa de possuir para que o *Actor* se possa movimentar. No caso de o *LocalStatus* não possuir os mapas necessários tem que os solicitar ao *GlobalCoordinator* antes de iniciar o *Actor*;
- `String stops` – é o nome dos mapas com os locais de paragem a que aquele ator está sujeito. Estas paragens podem ser semáforos, lojas, cafés, paragens de autocarro, comboio ou *tram*, sinais de *stop*, etc. Tal como os mapas das estradas para o movimento dos atores, o *LocalStatus* precisa de possuir estes mapas, caso contrário tem que os solicitar ao *GlobalCoordinator* antes de iniciar o *Actor*;

setActorStatus

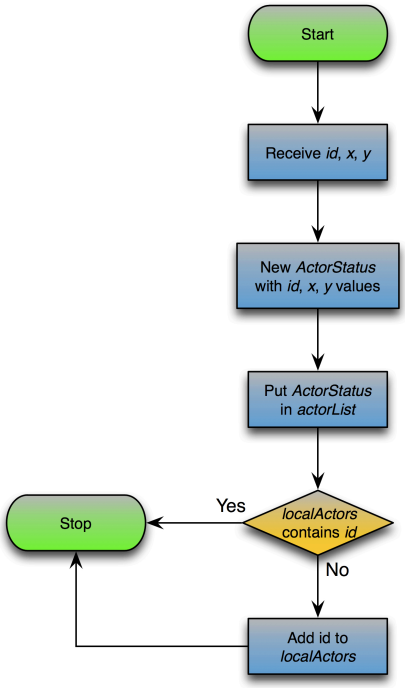

Figura 4.6 - Fluxograma do funcionamento da função *setActorStatus*

Como já foi dito anteriormente, a tabela em que todos os *Actors* estão a ser guardados não guarda o ator completo, mas apenas as informações necessárias de cada um deles. Para isso, existe o *ActorStatus*, um objeto que apenas guarda a informação relevante dos *Actors*. Neste momento definiu-se que as informações relevantes seriam o *id* e as suas coordenadas. Assim, esta função `setActorStatus` recebe como parâmetro as informações relevantes, coloca-as num *ActorStatus* e guarda-as na tabela `actorList`, como é perceptível no fluxograma apresentado na Figura 4.6. Como essa tabela tem como chave o *id* de cada *Actor*, basta que seja inserido um objeto com a mesma chave para que a anterior seja substituída, sendo essa a razão pela qual só é necessário que se insira na lista e não tenha que se retirar nenhum. Se um ator evoca esta função é porque está associado àquele *LocalStatus*, assim sendo de cada vez que a função é evocada faz-se a verificação se o seu *id* já está ou não inserido na lista dos *Actor* associados, a `localActors`. Na Figura 4.7 é apresentado um exemplo real em que um *Actor*, *Ped17*, faz uma atualização do seu estado ao *LocalStatus*.

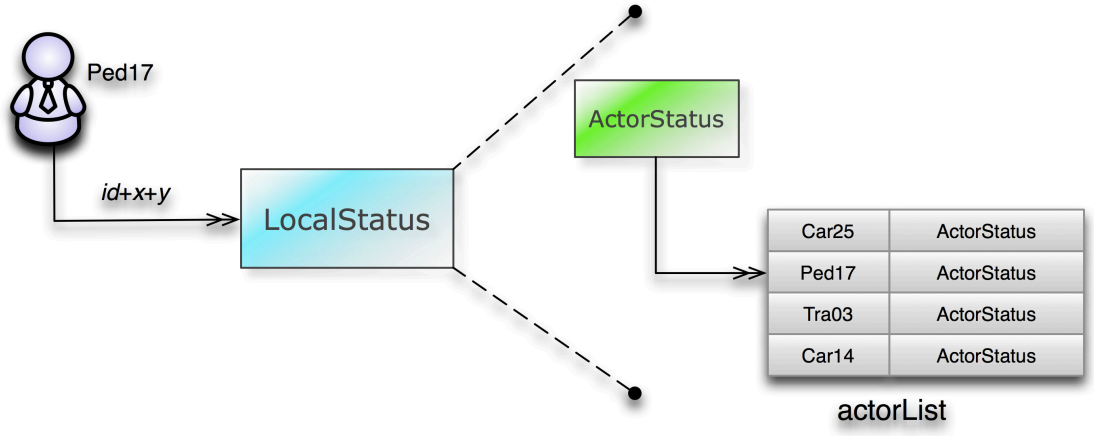

Figura 4.7 - Exemplo de funcionamento da função *setActorStatus* por parte do Ped17

getNeighbours

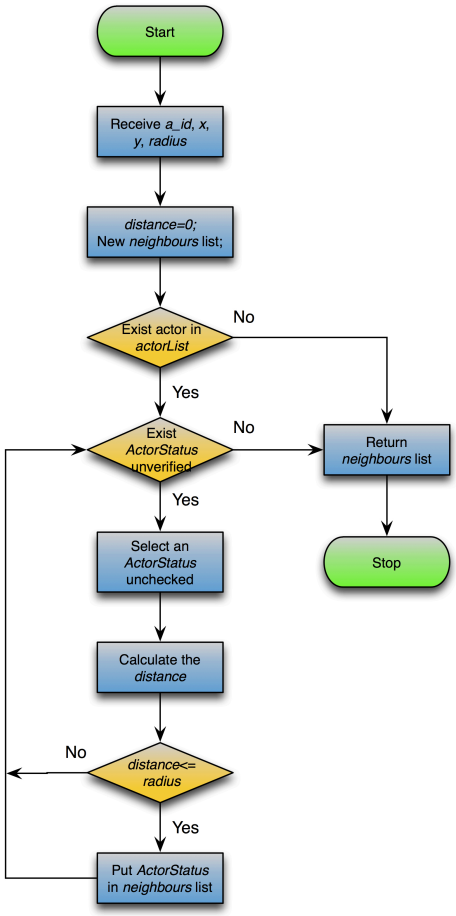

Figura 4.8 - Fluxograma demonstrador do funcionamento da função *getNeighbours*

Outra das funções implementadas foi a função `getNeighbours`, que foi implementada de forma a que sempre que é evocada devolve todos os vizinhos do nó que a evocou. Como cada ator tem necessidades diferentes no que diz respeito à área de procura de vizi-

nhos, o raio da circunferência em volta do nó, para pesquisa de vizinhos, é um parâmetro que tem que ser passado à função. É facilmente perceptível que um *Car*, que tem velocidades mais elevadas que *Ped*, necessite de um raio de vizinhança maior pois a sua aproximação é feita mais rapidamente. Assim, a cada movimentação, o *Actor* tem de evocar esta função de forma a conhecer a sua vizinhança para, em função disso, ajustar a sua reação. Pelo fluxograma apresentado na Figura 4.8, é perceptível que ao evocar a função, o ator tem que passar como parâmetro à função o seu *id*, `a_id`, a sua posição atual, `x` e `y`, e o raio de pesquisa, `radius` para que a procura seja iniciada. A fórmula usada para se saber se dois *Actors* são ou não vizinhos é a distância euclideana, apresentada de seguida:

$$distancia = \sqrt[2]{\left(x_i - x_f\right)^2 + \left(y_i - y_f\right)^2} \quad (1)$$

em que o x_i é a coordenada *x* do ator que evocou a função e x_f é a coordenada *x* do ator que se esta a testar se está na vizinhança, sendo aplicada a mesma associação para os valores de *y*. No caso de o valor da distância ser menor ou igual ao valor do raio passado por parâmetro, então o nó é considerado vizinho e adicionado à tabela de vizinhos a ser devolvida. Este procedimento é repetido para todos os atores que estejam ativos na simulação de forma a descobrir todos os vizinhos existentes. No final de percorrer toda a tabela dos *Actors*, todos aqueles que forem considerados vizinhos estão guardados numa outra tabela, com características semelhantes às da `actorList`, denominada de `neighbours`, como se pode ver na Figura 4.8. Essa tabela é retornada para o nó que evocou a função que fica assim a saber todos os nós que são seus vizinhos naquele momento. Para uma melhor percepção de como é efetuado na prática o algoritmo de procura, é apresentada a Figura 4.9 que demostra qual seria o resultado da evocação desta função por parte do ator *Car17*.

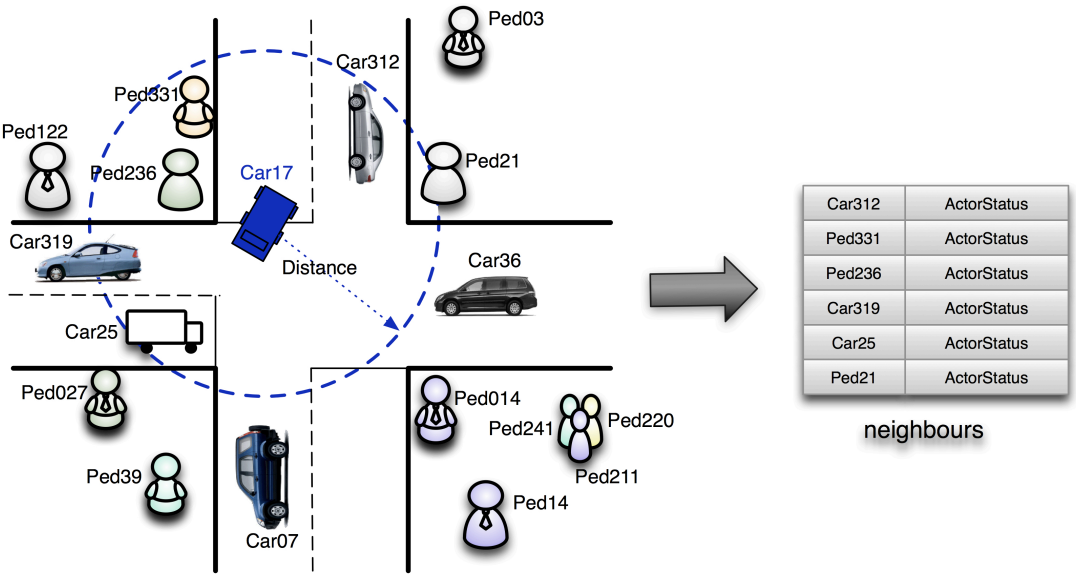

Figura 4.9 - Exemplo de procura de vizinhos do Car17

setListActorStatus

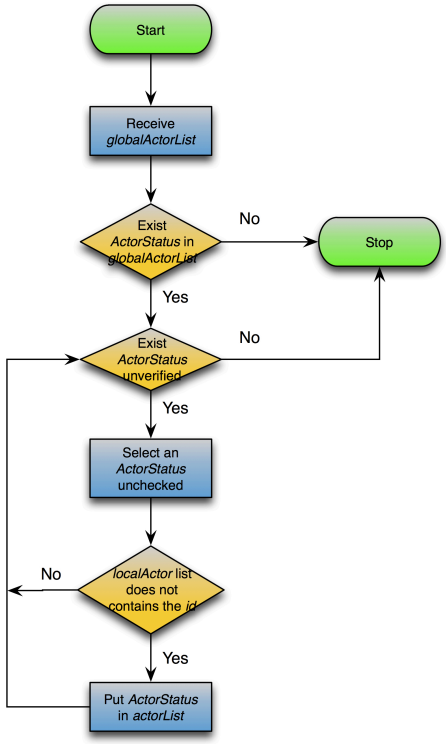

Figura 4.10 - Fluxograma da execução da função *setListActorStatus*

Por fim, foi implementada a função `setListActorStatus`, que coloca as atualizações que o *LocalStatus* vai recebendo na tabela dos atores. De cada vez que um *LocalStatus* envie uma atualização apenas envia as atualizações dos seus atores locais. Uma outra verificação que é feita é que a cada nova mensagens *multicast* que é recebida seja verificada a sua

origem para que não leia as mensagens que ele próprio enviou, de forma a assegurar que quando chegam as atualizações ao *LocalStatus* não há, por algum erro que tenha existido, uma desatualização dos seus *Actors*. Apesar disso, por cada ator que chega é feita uma comparação, verificando se o *id* esta contido na `localActors`. No caso de não estar, é atualizado o valor que já existia ou, no caso de ainda não estar inserido, é colocada uma nova entrada na tabela `actorList`. Caso seja um nó que está na `localActors`, é ignorado e passa-se ao próximo da tabela de atualização, caso ainda exista. O fluxograma mostrado na Figura 4.10 demostra o funcionamento que foi anteriormente explicado.

4.2.4 LocalCoordinator

Tal como acontece com o *GlobalCoordinator*, também o *LocalCoordinator* tem um ficheiro para inicializar todos os componentes que tem que ser ativos no respetivo computador. Da mesma maneira que o *GlobalCoordinator* foi implementado seguindo o que foi definido na secção 3.9.1.1, o mesmo acontece com o *LocalCoordinator*. No entanto, as etapas da inicialização do *LocalCoordinator* são ligeiramente diferentes do *GlobalCoordinator*, pois este tem diferente objetos associados a si, sendo a inicialização seguida para esta classe a que se encontra no secção 3.9.2.1.

Da mesma forma que o *GlobalCoordinator*, também o *LocalCoordinator* recorre ao auxílio de um ficheiro *properties* para que sejam carregados alguns valores necessários ao arranque da simulação.

4.2.4.1 Ficheiro properties

Apesar de também o *LocalCoordinator* usar as capacidades oferecidas pelo ficheiro *properties*, este não carrega uma quantidade de informação tão grande como o *GlobalStatus* pois, neste momento do desenvolvimento, o *LocalStatus* não necessita de tanta informação. No entanto, e tendo em conta o que já foi dito sobre os ficheiros deste tipo, a qualquer altura podem ser inseridas novas informações no ficheiro e usá-las no arranque do sistema.

4.2.5 Generator

A classe *Generator*, como já foi dito, funciona em separado do *GlobalStatus*, mesmo que na hora de criar um novo *Actor* para a simulação o tenha que fazer através dele. Assim, a classe *Generator* foi implementada como sendo uma *thread*, o que significa que tem

um funcionamento independente do que se esta a passar no sistema, embora esteja a trabalhar para ele.

4.2.5.1 Variáveis globais

Como já foi falado, os *Generators* são iniciados pelo *GlobalCoordinator* e os parâmetros são carregados a partir do respetivo ficheiro *properties*. Esses parâmetros são guardados na classe usando variáveis globais apresentadas no diagrama de blocos (Figura 4.1).

Já foi explicado anteriormente que o funcionamento dos *Generators* é baseado em aleatoriedade. As variáveis `top_lim` e `low_lim` são as que compõem o intervalo em que a variável aleatória precisa de acertar para que seja criado um novo nó no sistema, ou seja, se a variável aleatória for um valor dentro do intervalo dessas duas variáveis será criado um novo ator. Os valores da variável aleatória podem tomar valores entre 0 e 1, pelo que o resultado do pedido de um valor aleatório será um valor decimal. Já as variáveis `posi_x`, `posi_y`, e `speed` assumem também elas valores decimais, e são referentes aos valores que tem de ser passados aos novos *Actors*, sendo a `posi_x` a coordenada inicial em *x*, a `posi_y` a coordenada inicial em *y* e a `speed` o valor da velocidade inicial do novo ator. Todas estas variáveis serão de valor fixo e tem que ser passadas como sendo um parâmetro da classe no seu arranque.

Como foi dito, os nós são identificados com 3 caracteres, definido pelo tipo de nó, mais, no mínimo, 2 caracteres numéricos, o primeiro para identificar o *Generator* de origem e o segundo o número do nó. Assim, a variável, `send`, define o valor que é atribuído a cada novo *Actor* que é criado, sendo incrementada a cada novo *Actor* que é enviado para um dos *LocalStatus*.

Nas variáveis globais que se consideraram necessárias estão incluídas ainda mais três. A `gene_type` é a variável que dá o nome ao gerador, sendo esse nome constituído por 3 letras, que definem o tipo de atores que está a gerar, e um número, que é o que distingue os diferentes *Generators* do mesmo tipo. Assim, o nome que vai ser dado aos *Actors* criados por esse gerador é o seu próprio nome mais o valor da variável `send`. As outras duas variáveis são o nome dos mapas, `maps`, e das paragens, `stops`, associadas aos nós que são criados pelo gerador.

4.2.5.2 *Funções*

A classe `Generator` apenas possui duas funções, a `run` e a `new_actor`, sendo que a `new_actor` está a ser constantemente evocada pela função `run`. Uma vez que esta classe é `Thread` não vai haver nenhum outro objeto a evocar as suas funções, apenas ele as irá evocar.

É no método `run` que é implementado tudo o que a *thread* tem que fazer como processo independente. Uma vez iniciado este método, diz-se que a *thread* foi iniciada e é feita a execução de todo esse método, sendo que depois deixa de existir.

run

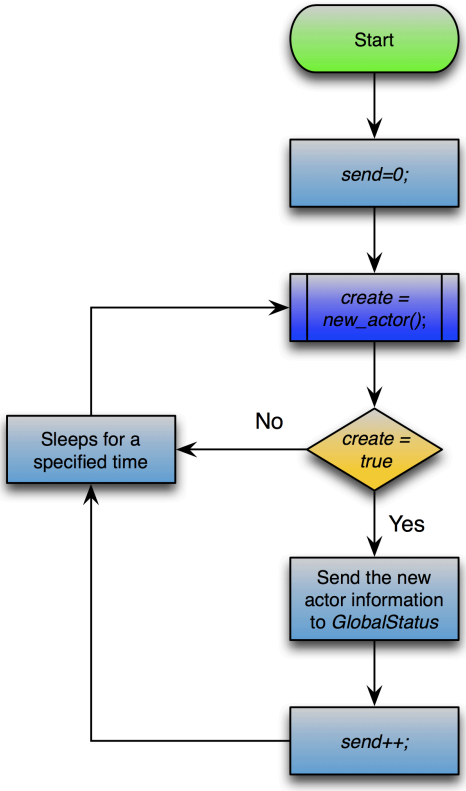

Figura 4.11 - Fluxograma descritivo do funcionamento da função *run*

Como se pode ver pelo fluxograma da Figura 4.11, o método `run` dos geradores apenas evoca a função `new_actor` e testa o que foi devolvido dessa evocação e, no caso de ser `true`, cria um novo *Actor*. Para que esta execução não seja feita apenas uma vez, dentro do método `run` foi colocado um ciclo em que a sua condição é sempre verdadeira, fazendo com que seja feito infinitas vezes, ou seja, até terminar a simulação. Para que a evocação da função `new_actor` não seja feita a um ritmo muito elevado, fazendo com que a probabilida-

de de criação de atores seja muito grande, a *thread* é adormecida durante um período de tempo a cada ciclo que é executado. O tempo que cada gerador fica adormecido vai diferir em função do tipo de atores que o gerador está a gerar.

new_actor

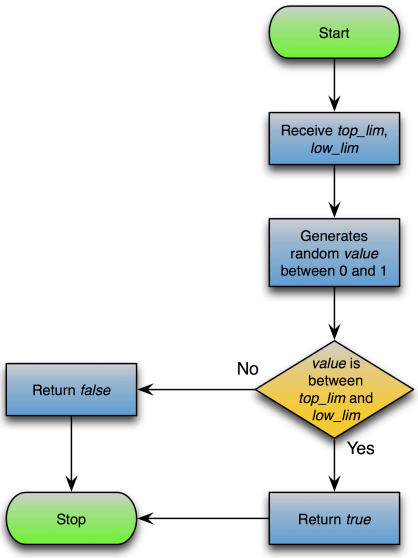

Figura 4.12 - Fluxograma do funcionamento da função new_actor

A função `new_actor` sempre que é evocada devolve `true` ou `false`, como se pode ver na Figura 4.12. Esta função apena gera um valor aleatório, entre 0 e 1, e verifica se está dentro dos limites definidos ou não. Em função disso responde com `true`, no caso de estar dentro do intervalo, ou `false`, no caso de estar fora desse intervalo. É em função da resposta devolvida por este método que a função `run` decide se cria ou não um novo ator para a simulação.

4.2.6 NetworkLogging

Para que seja possível haver um relatório do que se passa na LAN a que todos os computadores estão ligados, foi necessário implementar a classe `NetworkLogging`.

Esta classe, como já foi referido, será usada tanto pelo *GlobalCoordinator* como pelo *LocalCoordinator*, sendo que isso fará com que seja possível analisar a rede vista de diferentes formas, sendo também possível verificar se a troca de informação entre *GlobalCoordinator* e o *LocalCoordinator* está mesmo a acontecer.

4.2.6.1 Variáveis globais

Ao nível das variáveis globais, esta classe terá três, tanto no caso de estar a funcionar do lado do *GlobalCoordinator* como no caso do *LocalCoordinator*, como se pode verificar na Figura 4.1 e na Figura 4.2.

É através da variável `fLogger` que são evocados os diferentes tipos de *logs* padrão. Com o uso desta variável, basta apenas especificar o tipo de *log* que se pretende guardar para que fique guardado segundo o que é definido.

Esta variável, `typeOfLogging`, é a que define o tipo de *logging* que será feito durante a simulação. Como já foi dito, podem existir 3 tipos de *logging*, sendo este definido no início da simulação e mantido durante toda a simulação.

O `fh` é a variável que define o tipo de ficheiro usado para *logging*. Este tipo de ficheiro permite que a informação seja guardada de duas formas que, tal como o tipo de *logging*, tem que ser definido quando é declarada e não pode ser alterado durante o decorrer da simulação.

4.2.6.2 Funções

Esta classe tem apenas uma função que vai sendo evocada pelas outras classes, sempre que algum tipo de alteração de rede acontecer. Na Figura 4.13 é perceptível que o nível de informação que é guardado é definido inicialmente e mantido durante toda a simulação.

log

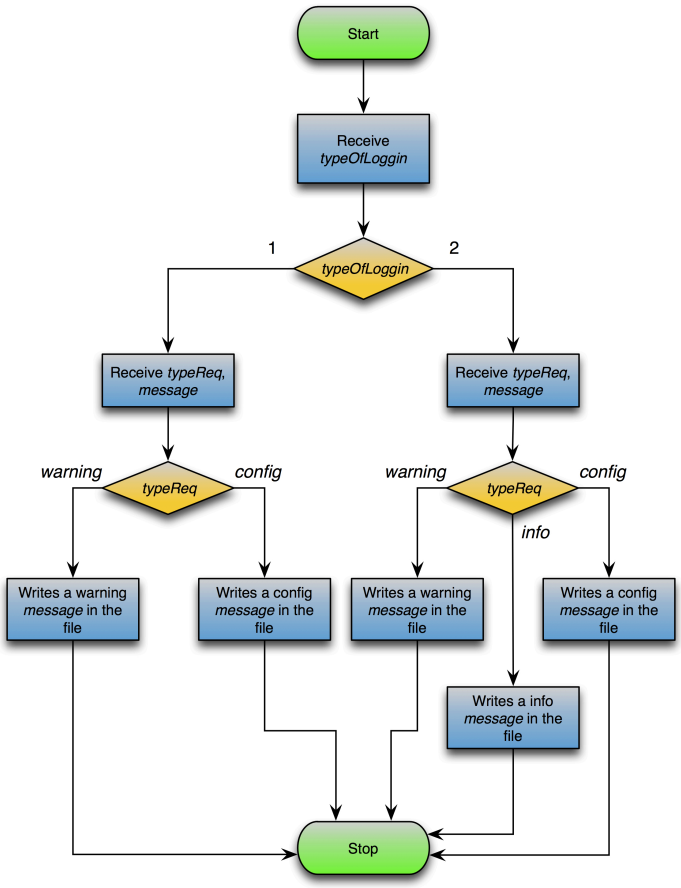

Figura 4.13 - Fluxograma com a descrição do funcionamento da função *log*

Esta função começa por definir o tipo de *logging* a ser usado, sendo que posteriormente sempre que é evocada, guarda o *log* juntamente com uma mensagem descritiva do que está a ser reportado da seguinte forma:

```
19/Jul/2011 16:56:05 um.simulator.core.NetworkLogging log
CONFIG: globalStatus ip 192.168.1.2 close tcp server conect on port 7575
with tcp client ip /192.168.1.3
```

4.3 Map

Este *package*, *um.simulator.map*, é aquele que é responsável por carregar e guardar os mapas. Para isso foram desenvolvidas 4 classe que não possuem qualquer função, com a exceção dos construtores e dos habituais *gets* e *sets*, pois a sua utilidade é guardar a informação dos mapas. Por esta razão, nenhuma classe deste *package* está presente nos diagramas de blocos apresentados (secção 4.1).

4.3.1 Global_Map

Como foi dito, esta classe é usada para se guardar a informação relativa aos mapas que são usados na simulação. Embora não exista nesta classe funções para serem evocadas por classes exteriores, é no construtor da mesma que é efetuado o carregamento da informação contida nos ficheiros dos mapas. Por cada ficheiro de mapas que tenha que ser carregado é criado um novo objeto `Global_Map`.

4.3.1.1 Variáveis Globais

No que às variáveis globais diz respeito, possui apenas duas e que são as listas com as informações relativas aos mapas. O que é guardado nestas tabelas são objetos definidos neste *package*.

Assim, foi implementada uma lista, denominada de `line_list`, para guardar todas as linhas presentes no ficheiro. Na sua composição, esta lista tem uma série de objetos do tipo *Map_Line* que é explicado mais adiante (secção 4.3.2).

A outra lista que foi implementada nesta classe, a lista `point_list`, é responsável por guardar todos pontos independentes do mapa carregado, ou seja, é composta por objetos do tipo *Map_Point*, também ele descrito mais à frente (secção 4.3.3).

4.3.1.2 Funções

Embora esta classe não possua qualquer tipo de função, o seu construtor funciona como sendo uma função que faz o carregamento dos mapas.

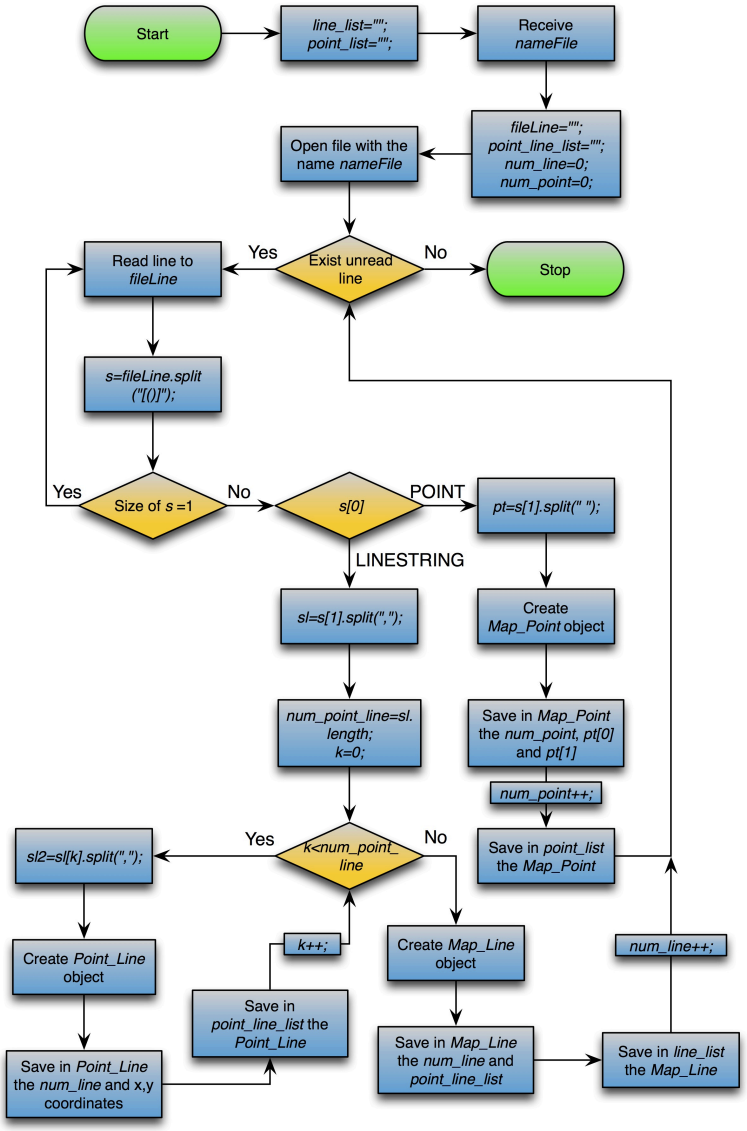

Figura 4.14 - Fluxograma do construtor da classe *Global_Map*

Como se pode ver pelo Figura 4.14, esta classe começa por ter de receber o nome do ficheiro que contem a informação a ser guardada. A leitura do ficheiro é feita linha a linha e a forma como é guardada a informação depende da informação lida, podendo ser guardada como sendo um ponto ou como como sendo uma linha, ou seja, um conjunto de pontos. Cada informação que é guardada neste objeto é também ela um objeto pertencente a este *package* dedicado apenas ao mapas.

4.3.2 Map_Line

Como foi dito a quando da explicação da classe *Global_Map* (secção 4.3.1), este objeto é onde é guardada cada linha do ficheiro que esta a ser carregado. Assim, cada entra-

da da lista `line_list` será um objeto deste tipo com a informação de todos os pontos constituintes da linha.

4.3.2.1 Variáveis Globais

Este objeto, ao nível de variáveis globais possui 2 para que seja definida cada linha do mapa. Como em cada ficheiro a ser carregado terá por certo mais que uma linha, para que não exista confusão entre as linhas, tem de existir uma variável que faça uma diferenciação, ou seja, que identifique cada uma das linhas. Essa variável foi denominada de `line_id` e consiste apenas num número inteiro que é passado sempre que é criado um novo objeto.

A outra variável que foi implementada foi uma lista, a `point_line`, que é responsável por guardar cada um dos pontos constituintes da lista. Como cada um dos pontos lidos do ficheiro será um objeto *Point_Line*, que será explicado mais adiante (secção 4.3.4), esta lista é constituída por objetos com a informação referente aos pontos.

4.3.2.2 Funções

Como já foi referenciado, nenhuma das classes deste *package* possui qualquer função que não seja as habituais funções de *get* e *set* das variáveis globais.

4.3.3 Map_Point

Uma vez que existe um objeto para guardar cada linha que exista no ficheiro que está a ler, existe também um objeto para que seja guardado cada ponto individual que exista nesse mesmo ficheiro.

4.3.3.1 Variáveis Globais

Nesta classe houve a necessidade de serem implementadas 3 variáveis globais. Como o ponto é a unidade basilar de um mapa, não há necessidade de haver qualquer tipo de lista para guardar outros objetos. Assim, como há a necessidade de se distinguir os diferentes pontos que possam existir num ficheiro, foi criada a variável `point_id` que é um número inteiro diferente em cada objeto `Map_Point`. No entanto, quando se inicia uma novo *Global_Map*, ou seja, um novo ficheiro de mapas, esta contagem é reiniciada.

As outras duas variáveis que foram implementadas nesta classe dizem respeito às coordenadas do ponto que está a ser guardado. Assim, foi implementada a variável `point_x` para a coordenada *x* e a variável `point_y` para a coordenada em *y* de cada ponto.

4.3.3.2 Funções

Esta classe, como todas as deste *package*, não possui qualquer tipo de função adicional às conhecidas *get* e *set*.

4.3.4 Point_Line

Para finalizar a apresentação das classes do *package map*, será apresentada a classe `Point_Line` que foi implementada para que albergasse a informação dos pontos pertencentes às linhas dos mapas.

4.3.4.1 Variáveis Globais

Representando esta classe a unidade simples que é um ponto num mapa, tal como a *Map_Point*, ambas as classes são muito semelhantes na sua implementação. Assim, nesta classe foram implementadas também 3 variáveis. Como esta classe representa um ponto de uma linha, achou-se importante que cada um destes nós possuíssem uma identificação da linha a que pertence. Assim, foi implementada uma variável denominada de `line_id` que terá o valor que a variável com o mesmo nome da classe *Map_Line*.

Qualquer que seja o ponto que esteja a ser representado tem de ter pelo menos duas coordenada, pelo que nesta classe foram implementadas as variáveis `point_x` e `point_y` para se guardarem as coordenadas em *x* as coordenadas em *y*, respectivamente.

4.3.4.2 Funções

Sendo esta mais uma classe pertencente ao *package map*, também nela não foram implementadas quais quer funções adicionais.

4.4 Communications

O *package um.simulator.communications*, denominado apenas de *communications* de agora em diante, é onde se encontram todas as classe relativas às comunicações entre as entidades.

4.4.1 Multicast

A quando da análise do funcionamento do sistema foi decidido que se iria usar o *multicast* IP para a difusão da informação. Como ponto de partida para a implementação foi necessário fazer um diagrama com a sequência de eventos que o *multicast* gera, para que partindo daí se consiga estabelecer que tipo de mensagens necessárias.

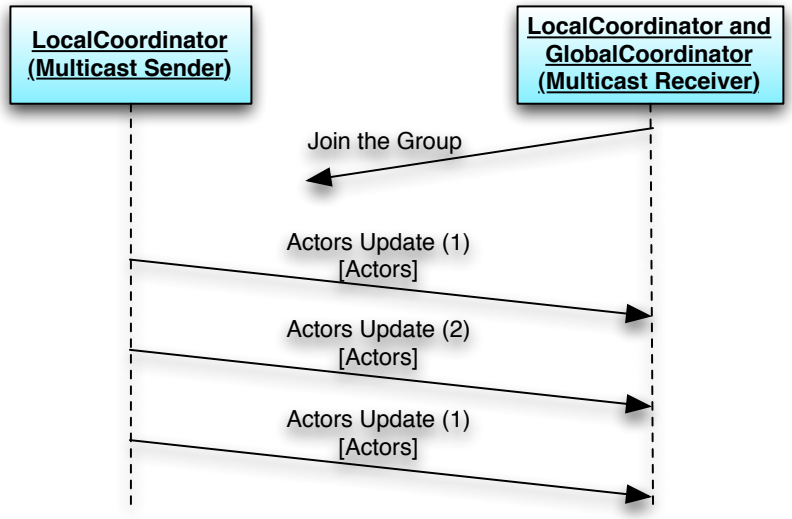

Figura 4.15 - Digrama de sequência do *multicast*

Na Figura 4.15 pode-se ver a sequência de acontecimentos relacionados com o *multicast* que acontecerão no sistema a ser desenvolvido. O primeiro evento será os *LocalCoordinators*, o *GlobalCoordinator* e a *Visualization* fazerem *Join* ao grupo *multicast* destinado ao simulador. A partir desse momento qualquer *LocalCoordinator* que já tenha *Actors* associados a si pode começar a enviar as suas atualizações para o grupo, fazendo com que essa informação chegue a todas as entidades que fizeram o *Join*.

Quando um dispositivo faz *Join* a um grupo *multicast* fica habilitado tanto a enviar como a receber mensagens endereçados a esse grupo. Assim, sempre que um *LocalCoordinator* envia uma mensagem para o grupo receberá também a sua própria mensagem, pois o seu endereço faz parte do grupo. No entanto, para que não perca tempo de processamento a analisar informação que foi ele que enviou, antes de começar a processar a informação de uma mensagem recebida, o *LocalCoordinator* vai verificar se foi ele quem enviou aquela mensagem para a rede, logo vai existir a necessidade de ter em todos os tipos de mensagens o IP de origem.

4.4.1.1 Formatação das mensagens

Para que as várias entidades saibam que informação vai em cada mensagem e qual o formato da informação, tiveram de ser estabelecidas normas para as mensagens que serão enviados. Como foi dito anteriormente, o *payload* máximo dos datagramas *multicast* é de 1472 bytes, para que não ocorra fragmentação na origem, pelo que se teve de partir deste valor de referência para arquitetar as mensagens.

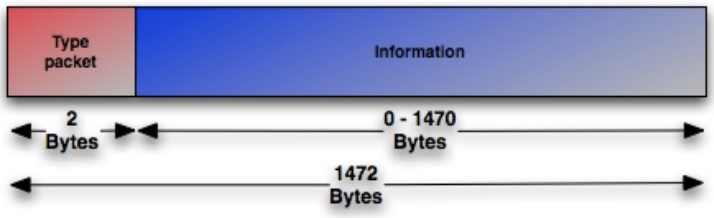

Figura 4.16 - Formato geral das mensagens e enviar por *multicast*

A primeira divisão que se fez à mensagem foi criar um campo que define o tipos de mensagem, pois podem existir mensagens com diferentes tipos de informação. Assim, e como se pode ver na Figura 4.16, os 1472 bytes foram divididos da seguinte forma: 2 bytes para o tipo de mensagem; 1470 bytes para a informação a ser enviada.

Tentando maximizar a quantidade de informação enviada, chegou-se à conclusão que o tamanho mínimo aceitável para o campo *Type packet* são 2 bytes, sendo compostos por 2 caracteres decimais. Mesmo não sendo até momento usadas todas as possibilidade de tipos de mensagens achou-se por bem dar uma margem de intervalo para o caso de necessidades futuras.

A quando do desenho dos tipos de mensagens necessários ao *multicast* surgiram 2 tipos, embora não difiram muito entre eles.

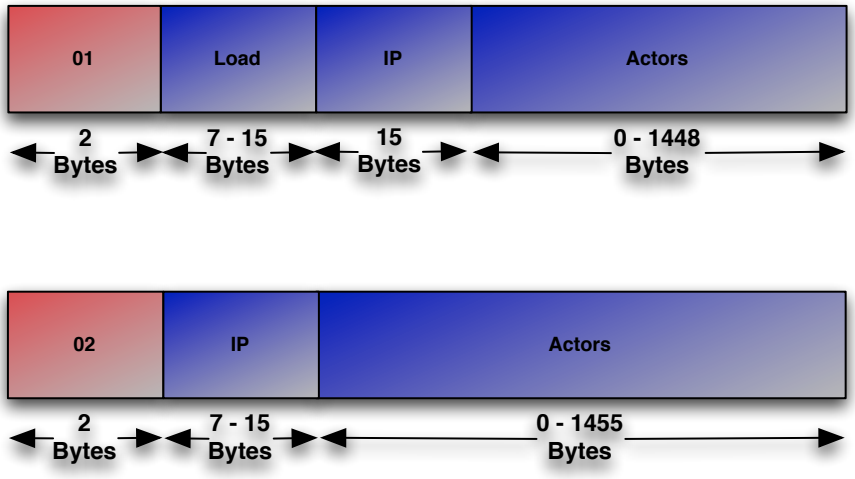

Figura 4.17 - Tipo de mensagens a enviar por *multicast*

As mensagens do tipo "01" são as mensagens que os *LocalCoordinators* enviam com a primeira série de atores da sua lista. Junto com o tipo de mensagens, usado para o destino saber o formato dos dados, vai o IP de origem da informação, para que o *GlobalCoordinator* consiga associar a carga ao devido *LocalCoordinator* e também para que os *LocalCoordinators* consigam saber se foi ele quem enviou aquela atualização, e também a carga do emissor no momento do início do envio, para que o *GlobalStatus* consiga distribuir a carga entre os *LocalCoordinators*.

No caso de o *LocalCoordinator* emissor ter um número de atores associados a si que não caibam todos dentro do primeiro datagrama, mensagem do tipo "01", os restantes atores serão enviados em mensagens do tipo "02". Este tipo de mensagem não inclui o campo *Load*, com a carga do emissor, pois a carga só será enviada de cada vez que o *LocalCoordinator* recomeçar a enviar a sua lista de atores, ou seja, nas mensagens do primeiro tipo.

4.4.1.2 MulticastSender

A primeira classe que teve de ser desenvolvida foi a *MulticastSender* que é a classe responsável por elaborar e enviar as mensagens com as atualizações para o grupo *multicast*.

Como a classe `MulticastSender` é um processo que tem que estar sempre em funcionamento, tal como acontece com os *Actors*, pelo que teve que ser implementado como sendo `Thread`. Isto acontece porque o *GlobalStatus* e os restantes *LocalStatus* têm que ser constantemente atualizados com as novas posições dos atores.

4.4.1.2.1 Variáveis Globais

Esta classe, presente apenas nos *LocalCoordinators* (Figura 4.2), implicou que tivessem que ser criadas 4 variáveis globais, que são utilizadas para que a informação esteja disponível em qualquer altura da execução.

A variável `listToSend` é uma cópia da tabela de todos nós do *LocalCoordinator* que é pedida a cada nova atualização que tem que ser enviada. É usada uma cópia integral da `actorList` para que esta fique operacional enquanto é percorrida, ator a ator, para ser enviada para o grupo *multicast*. Isto vai fazer com que se um *Actor* se movimentar enquanto a tabela esta a ser enviada pode guardar logo a sua nova posição e não ficar à espera que a tabela termine de ser envia para depois a atualizar.

Para que não sejam enviadas para o grupo *multicast* informações que sejam conflituosas e contraditórias, cada *LocalCoordinator* envia para o grupo apenas as informações dos *Actors* que tem associados a si. Para que isso seja possível, tem que ser feito um *merge* entre a `actorList` e a `localActors` e enviar apenas os nós cujo o *id* consta na `localActors`. Pela mesma razão que é feita uma cópia integral da `actorList`, também foi necessário criar uma lista para albergar uma cópia da `localActors` para que não fique inoperacional durante o tempo que é usado para o envio das atualizações. Esta cópia vai garantir que no caso de ser criado um novo ator enquanto está a ser preparado uma mensagem para a atualização, este não vai ficar de fora da lista de *Actors* associados.

A variável `packet_type` é o tipo de mensagem que vai ser enviado, já o `ipLocal` é usado para guardar o IP de origem da mensagem que se está a preparar para enviar.

4.4.1.2.2 Funções

Uma vez que a classe `MulticastSender` é uma `Thread`, como já foi dito anterior mente, tem que possuir o método `run`. Devido as necessidade desta classe, esta foi a única função que foi implementada e o seu funcionamento é descrito pelo fluxograma exibido na Figura 4.18.

run

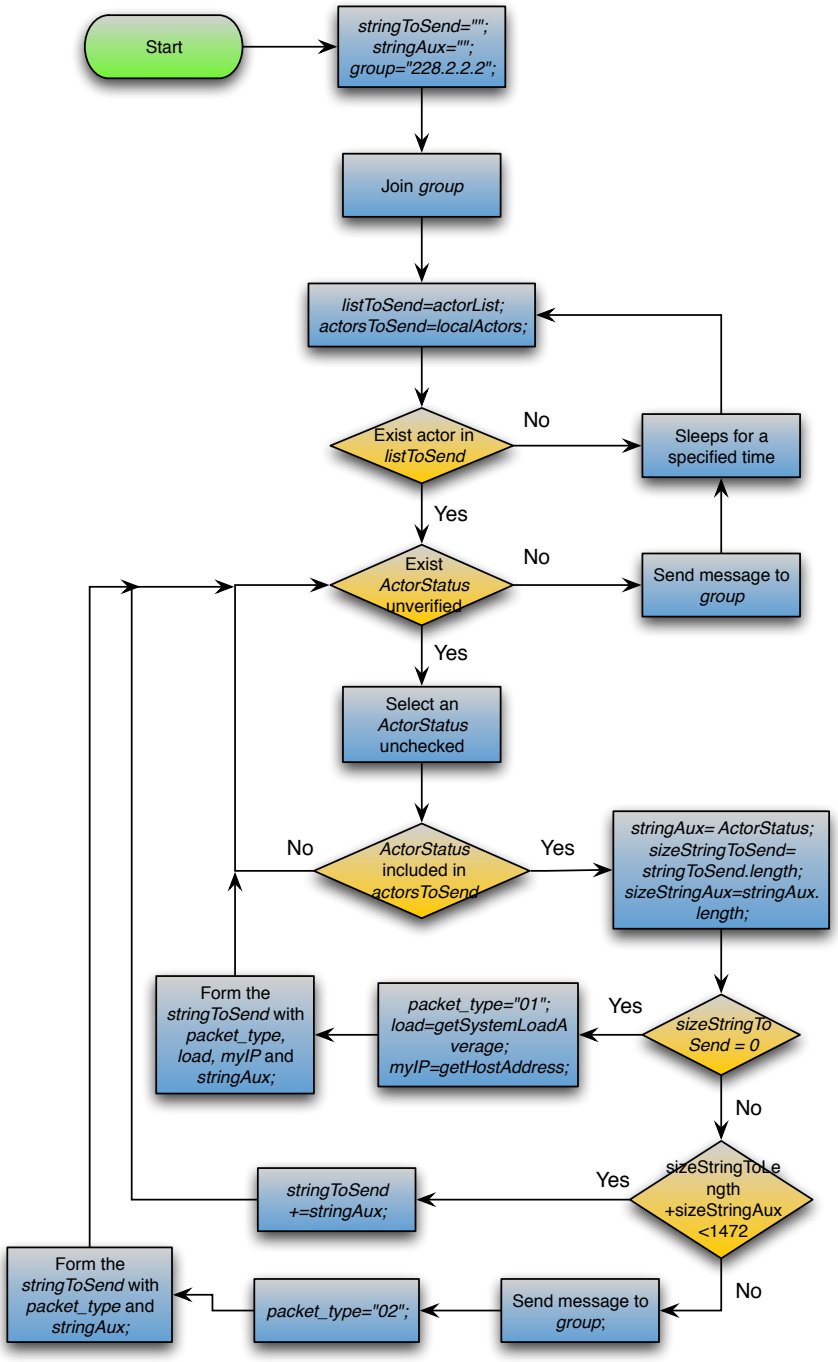

Figura 4.18 - Fluxograma do funcionamento do método *run*

Nesta função, que é a única da classe, é feito todo o processo da atualização dos atores, desde a construção das mensagens até ao seu envio para o grupo *multicast*. É importante referir que uma vez que toda a informação que consta nas mensagens é do tipo `String`, houve a necessidade de se incluírem caracteres especiais para separar os diversos campos

que constituem a mensagem. Assim, para separar os vários componentes dos *Actors* é usado o caractere "#" e para separar os vários campos dos mensagens, inclusive os vários *Actors*, é usado o caractere "@". Já para separar a carga do IP é usado o caractere "/". Assim sendo, um exemplo de uma mensagens poderia ser o seguinte:

```
01@0.001/192.168.231.152@Car02#5.236321#10.1267823#1264774229@Ped26#23.98
76231#22.867891#13498712@
```

em que seria uma mensagem do tipo "01", que no campo carga leva a carga e o IP do *LocalCoordinator* de origem, separados pelo "/", e envia para o grupo *multicast* a informação dos *Actors* "Car02" e "Ped26". Como também já foi referido anteriormente, as atualizações não vão ser enviadas constantemente para a rede, para que não haja um grande carga na rede e porque poderia haver o caso de não haver qualquer atualização a reportar e estar-se-ia a enviar informação repetida. Para que isso não aconteça foi colocado um *sleep* para que a `Thread` adormeça algum tempo de cada vez que faz uma atualização para o grupo. A Figura 4.19 elucida a forma como é feito o *merge* mas duas listas existentes de forma a serem encontrados os nós a enviar.

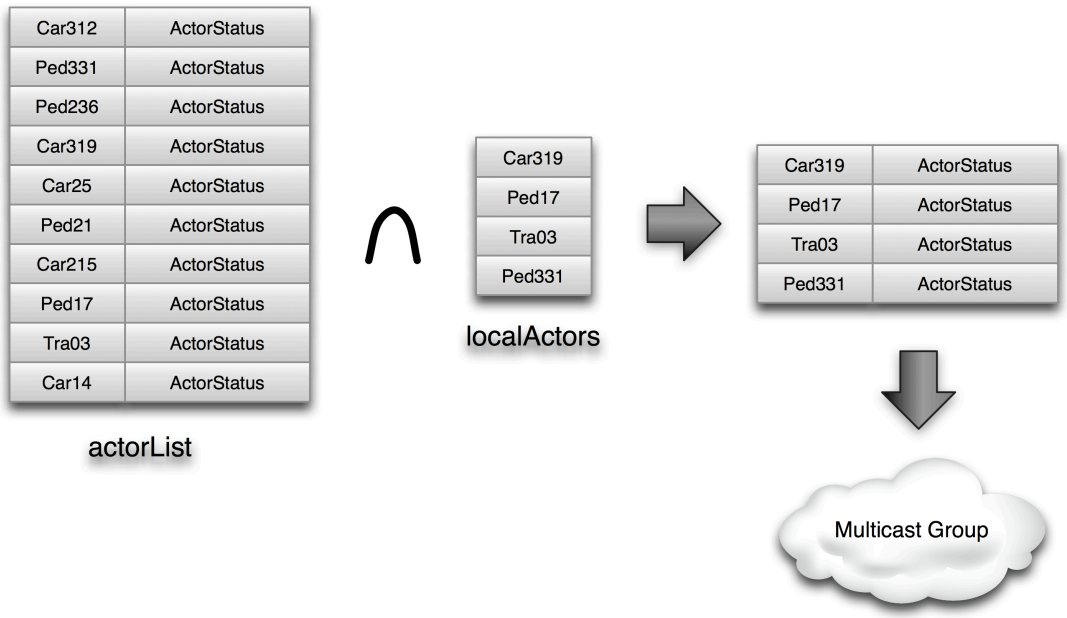

Figura 4.19 - Exemplo do envio de atualização para o grupo *multicast*

A quando da explicação do funcionamento do *multicast* no BartUM Simulator, foi referido que exista uma limitação relacionada com o tamanho das tramas. É referido nessa altura que seria implementada uma solução que consistia em encher as mensagens com o

máximo de informação e envia-las. A implementação dessa solução seguiu o seguinte algoritmo:

```
Enquanto há atores na lista{
          Se tamanho da mensagem + 1 ator <= 1472{
               Insere ator no mensagem
          }
          Se não{
               Envia mensagem atual
               Criar nova mensagem
               Insere ator na mensagem
          }
     }
```

Com este algoritmo fica garantido que as mensagens levam o máximo de atores possível e que no caso de perda de uma mesnagem apenas se perca informação relativa a um pequeno número de atores que será compensada na próxima atualização, como era pretendido.

4.4.1.3 MulticastReceiver

Em conjunto com a classe *MulticastSender* teve que ser implementada a classe que recebesse a informação que era enviada, a *MulticastReceiver*. Esta classe é responsável por estar sempre à escuta no grupo *multicast* para receber as atualizações.

A classe `MulticastReceiver` é também ela implementada como sendo uma `Thread` para que possa estar a escutar o grupo *multicast* sem que influencie o decorrer da simulação.

Como esta classe está presente nos três modos de funcionamento do simulador, *GlobalCoordinator*, *LocalCoordinator* e *Visualization*, também ela tem de ter três diferentes modos de operar, pois cada um deles precisa de retirar diferentes tipo de informação das mensagens recebidas.

4.4.1.3.1 Variáveis Globais

Como variáveis globais, esta classe apenas apresenta uma, a `receiverType`, sendo lá que é especificado que modo de funcionamento é que a classe vai estar associada. Assim, esta variável apenas assume 3 valores: `"localstatus"`, `"globalstatus"` e `"visualiza-`

tion". Para que a informação seja correta, é a classe *main* que, a quando da evocação do construtor, lhe passa como parâmetro a respectiva informação.

4.4.1.3.2 Funções

Tal como acontece com todas as classe que são implementados como *thread*, a classe `MulticastReceiver` tem de possuir uma função `run` para que a *thread* seja iniciada. Uma vez que o funcionamento desta classe é cíclico, é necessária apenas uma função, `run`. No fluxograma da Figura 4.20 é demonstrado o funcionamento deste método nas 3 diferentes possibilidades.

run

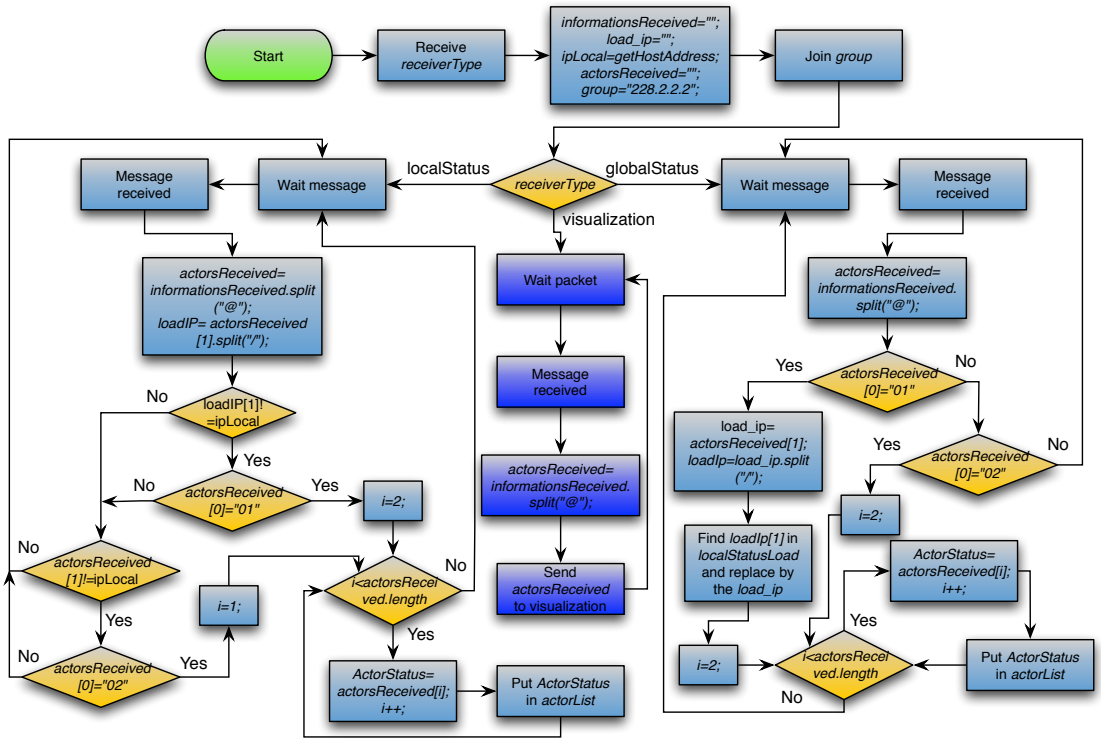

Figura 4.20 - Fluxograma do funcionamento do método *run*

Como se pode ver no fluxograma (Figura 4.20), o método `run` tem que optar por um dos modos de funcionamento e a partir desse momento fica sempre nesse tipo de funcionamento até ao fim da simulação. Isso é garantido um vez que após a decisão de qual o modo de funcionamento, o método fica "preso" num ciclo em que a condição de teste é infinitamente verdadeira. Durante toda a simulação esta função vai estar a receber as diversas atualizações que vão sendo enviadas ao grupo *multicast*. No caso de já ter informação sobre os *Actors* que chegam, esta função faz uma atualização da informação que tinha, caso contrário

acrescenta os novos atores à sua lista. Os três modos de funcionamento têm quer ser diferenciados pois em cada um dos modos o tratamento dado à informação recebida é diferente. No caso do *LocalCoordinator* tem que tem em consideração a origem da mensagem, parâmetro que é indiferente aos outros dois modos de funcionamento. Já no caso do *GlobalCoordinator*, não há qualquer seletividade na origem da informação mas tem que ter em conta a informação referente à carga do *LocalCoordinator* de origem. Por fim, a *Visualization* apenas se importa em receber os *Actors* para que possa representar o seu movimento, não tendo qualquer atenção à origem das mensagens ou à carga dos *LocalCoordinator*. A Figura 4.21 permite que se visualize como serão feitas as inclusões das atualizações dos atores que vão chegando.

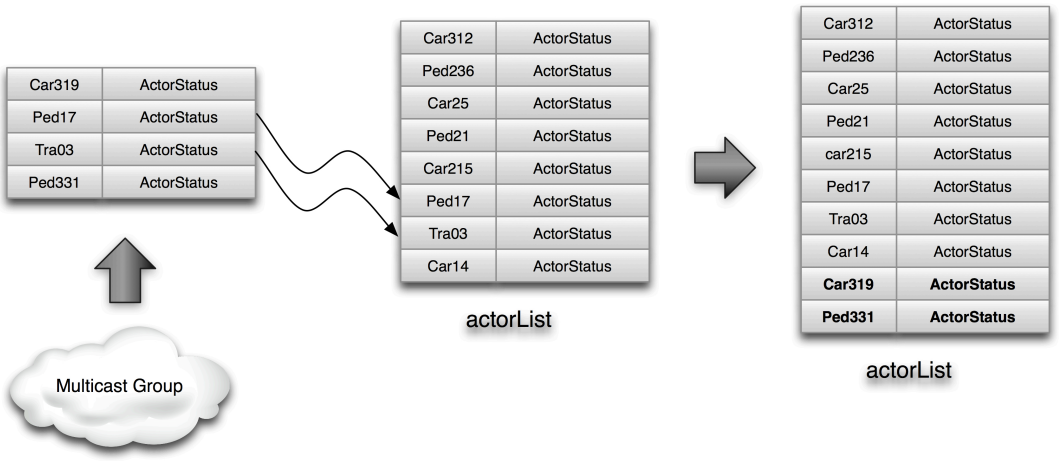

Figura 4.21 - Exemplo da recepção de uma atualização

Sempre que a classe recebe uma mensagem com uma nova atualização, a mensagem é decomposta nos diferentes campos que o constituem, e em função do seu tipo, `"01"` ou `"02"`, a informação é tratada da maneira devida, como se pode ver na Figura 4.20, pois a sua composição (Figura 4.17) é também ela diferente.

4.4.2 TCP

Na análise do BartUM Simulator ficou estabelecido que juntamente com o *multicast* IP, não confirmado, iria ser usado um protocolo confirmado para comunicações em que o conteúdo precisa de ser assegurado, sendo o protocolo escolhido o TCP. Tal como no *multicast*, também para o TCP foi utilizado um diagrama de sequência para a análise das necessidades do protocolo.

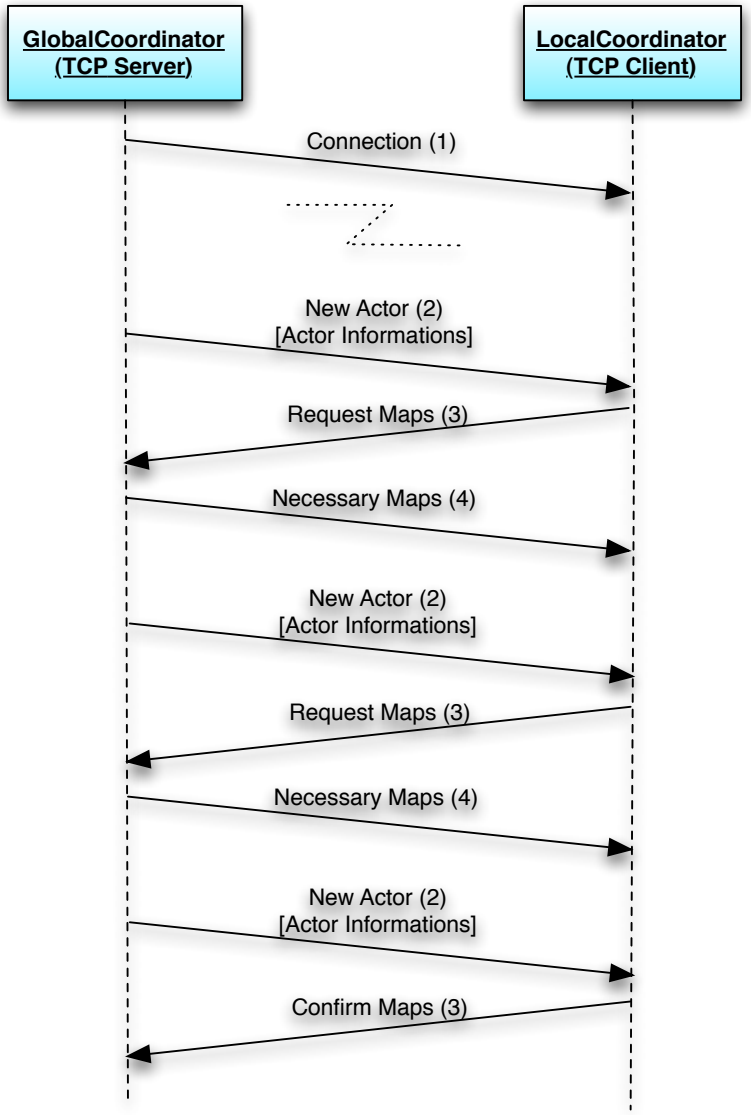

Figura 4.22 - Diagrama de sequência TCP

No diagrama de sequência representado na Figura 4.22 pode-se ver o procedimento que é feito desde o início da conexão até à troca de dados. Assim, após a obtenção da conexão entre o cliente (*LocalCoordinator*) e o servidor (*GlobalCoordinator*) a primeira coisa a ser feita é informar o cliente de qual a porta em que se terá de ligar para comunicar com o seu servidor *slave*. Uma vez consumada a ligação entre o cliente e o servidor *slave*, começa a troca de informação entre eles. Como é perceptível, são enviadas ordens, do servidor para o cliente, para fazer nascer os atores e o cliente, após a verificação dos mapas, informa se necessita de novos mapas ou não. Na Figura 4.22, para os dois primeiros *Actors* o *LocalCoordinator* não possui os mapas, tendo que os pedir ao *GlobalCoordinator*, e no terceiro ator que recebe o *LocalCoordinator* já tem os mapas necessários aquele *Actor* pelo que apenas

confirma isso ao *GlobalCoordinator*. O funcionamento deste protocolo será sempre este no decorrer da simulação.

4.4.2.1 Formatação das mensagens

Fixados os cenários em que haverá necessidade de se usar este protocolo, foram especificados as mensagens que são usados para as trocas de informação necessárias, ou seja, estabelecer o protocolo de comunicação entre as entidades.

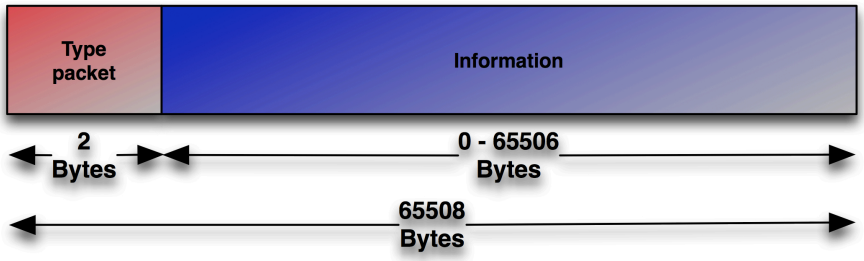

Figura 4.23 - Formato geral das mensagens a enviar por TCP

Uma primeira divisão feita à mensagem teve a ver com a necessidade de se criar um campo que definisse o tipo de mensagem, visto que no mesmo protocolo é viável que exista mensagens com diferentes tipos de informação, tal como acontece com o *multicast*. Cada mensagem é dividida em duas partes (Figura 4.23): 2 *bytes* para o tipo de mensagens, seguidas do corpo da mensagem.

O campo *Type packet* foi definido com 2 bytes pois achou-se que será o tamanho mínimo necessário para um sistema como o que será desenvolvido, que embora ainda não use todos os tipos disponíveis, deixa uma margem de possível progressão. Estes 2 bytes serão 2 caracteres decimais.

Sendo a informação tocada através deste protocolo muito diferenciada, projetou-se que seriam usados 4 tipos diferentes para a realização das trocas de informação.

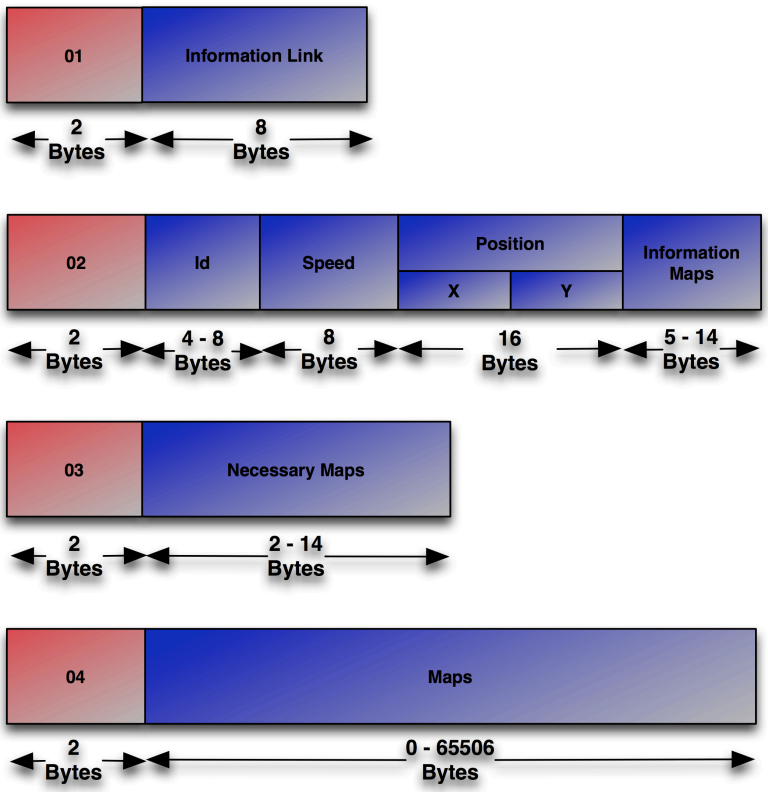

Figura 4.24 - Tipos de mensagens a enviar pelo TCP

As mensagens que forem enviadas com o valor "01" no *Type packet* serão mensagens que levarão informação sobre a ligação a fazer-se. Este tipo de mensagem existirá porque o *GlobalCoordinator* terá um servidor TPC ligado a uma porta, bem conhecida, à espera de todas a ligações TCP pretendidas. De cada vez que uma nova ligação é pedida, esse servidor criará um outro servidor TCP, que funcionará como *slave*, que atenderá apenas aquela entidade. Assim sendo, sempre que uma entidade solicita uma ligação ao servidor *master*, ser-lhe-á enviado uma mensagem que no campo o *Type packet* terá o valor "01" indicando a porta à qual se deve conectar para ter contacto direto com o seu servidor *slave*.

O segundo tipo de mensagens TCP, ou seja, os que tem o *Type packet* preenchido com o valor "02", são as mensagens que são enviados do *GlobalCoordinator* para o *LocalCoordinator* com as informações necessárias a criar um novo *Actor*. Nas mensagens deste tipo, representado na Figura 4.24, apenas estão representados os campos genéricos e comuns a todos os *Actors*, no entanto, dependendo do tipo de ator que é criado, podem ser enviadas mais ou menos informações. Assim, as informações que serão comuns a todos os

tipos de *Actors* são: o seu *id*; a sua velocidade; a sua posição, composta pela coordenada em *x* e coordenada em *y*; o nome dos mapas que ele necessita para o se movimentar.

Uma vez que cada ator que é mandado criar tem a informação de que mapas necessita para as suas movimentações, o *LocalCoordinator* que o alojará tem que garantir ao seu *Actor* que está na posse dos mesmos. Para garantir isso, de cada vez que uma mensagem com a ordem de criar um novo ator for recebida há a necessidade de se proceder à verificação da existência ou não desse mapa antes do *Actor* ser efectivamente criado. Para isso, foi especificado o *Type packet* "03", que é enviado do *LocalCoordinator* para o *GlobalCoordinator*, de cada vez que este lhe envia uma mensagem de tipo "02", com a informação de que mapas o *LocalCoordinator* necessita ou então apenas a confirmar que possui os mapas necessários.

Havendo a necessidade de haver trocas de mapas durante o decorrer da simulação, isso será feito através das mensagens que tenham como valor de *Type packet* o número "04". Este tipo de mensagem é a resposta às mensagens de tipo "03" no caso de o *LocalCoordinator* necessitar de algum mapa.

Sendo que o servidor TCP *master* está do lado do *GlobalCoordinator* e que apenas este tem, garantidamente, todos mapas existentes na simulação, no caso de haver uma entidade *Visualization* que se queira ligar à simulação terá de se registar no servidor *master*, tal como um *LocalCoordinator*, para que ocorra a criação um servidor *slave* de forma a que lhe sejam enviados todos os mapas. Para este efeito serão usados também o tipo de mensagem "04", uma vez que o tipo de informação a enviar é o mesmo.

O uso do protocolo TCP pressupõe que há um estabelecimento de conexão entre o servidor e o cliente. Neste caso, o estabelecimento da conexão será feito com um servidor que será o *master*, pois estará numa porta bem conhecida, que criará um outro servidor que será o *slave*, pois apenas estará dedicado aquela conexão e só existirá enquanto a ligação existir. Isto fará com que a conexão possa ser feita com clientes de diferentes tipos, neste caso com *LocalStatus* e com *Visualizations*.

4.4.2.2 TCPClient

O protocolo TCP pressupõe um funcionamento nos moldes de cliente - servidor. Para isso, teve de ser implementado uma classe para funcionar como cliente, a classe `TCPClient`.

Tal como todas as outras interfaces de rede, a classe `TCPClient` teve que ser implementado como uma `Thread` para ter um funcionamento autónomo. Esta classe permite ainda que haja um acesso à sua informação por outras classes de forma a terem acesso a uma interface de comunicação confirmada e também para ter acesso a informação que chega por essa mesma interface.

4.4.2.2.1 Variáveis Globais

No que diz respeito às variáveis globais, a classe `TCPClient`, que apenas está presente no *LocalCoordinator* (Figura 4.2) e na *Visualization*, tem um grande número delas uma vez que é a partir desta interface que é recebida a ordem de nascimentos de novos *Actors*.

Uma informação que é relevante na comunicação via TCP é a porta pela qual se está a comunicar. Nesta classe, a variável que guarda esta informação é a `port`, sendo essa informação um valor numérico.

Tendo em consideração que esta classe lida com muitas mensagens, entre enviar e receber, era expectável que tivesse várias variáveis relacionadas com as mensagens e seus conteúdos. Tal como acontece nas outras classe do *package communication*, é implementado o *id* dos atores, `id`, assim como os vários IP necessários, neste caso o IP do servidor com que esta a comunicar, `ipServer`, e o seu próprio IP, `ipLocal`. Tal como acontece com a classe *MulticastReceiver*, também esta classe tem dois tipos de funcionamento, *LocalCoordinator* e *Visualization*. A variável que vai diferir esse tipo de funcionamento é a `type`, que recebe essa informação quando a classe é iniciada. As outras variáveis que também estão relacionadas às mensagens são o tipo de mensagem, `packet_type`, a mensagem que é recebido e que é enviado, `packet_message`, os nomes dos mapas necessários ao *Actor* que está prestes a nascer, `maps`, e os mapas dos pontos de paragem desse mesmo *Actor*, `stops`. Outra variável que é importante no contexto da classe é o `packet_message_aux` que é usado para dividir a mensagem nos diferentes campos. Assim, sendo feita a divisão do `packet_message` pelo caractere "@", cada uma das entradas será um campo da mensagem. Por exemplo, depois da divisão, o entrada `pacet_message_aux[0]` será o valor do tipo de mensagem.

Além do *id* e dos nomes dos mapas, é necessário que se guarde os valores das posições inicias do nó que se prepara para nascer, o `position_x` para a posição em *x*, e o `position_y` para a posição em *y*, assim como o valor da sua velocidade inicial, `speed`.

4.4.2.2.2 Funções

Como as restantes classe do *communiction*, também o `TCPClient` tem como função apenas o `run`. Esta classe tem dois modos de funcionamento que varia consoante o ambiente onde é colocada a funcionar, se num *LocalCoordinator* se na *Visualization*. Pelo fluxograma da Figura 4.25, é perceptível que no caso de a classe estar inserida no *LocalCoordinator* terá um funcionamento interrupto, o que não acontece no caso de ser um cliente na *Visualization*.

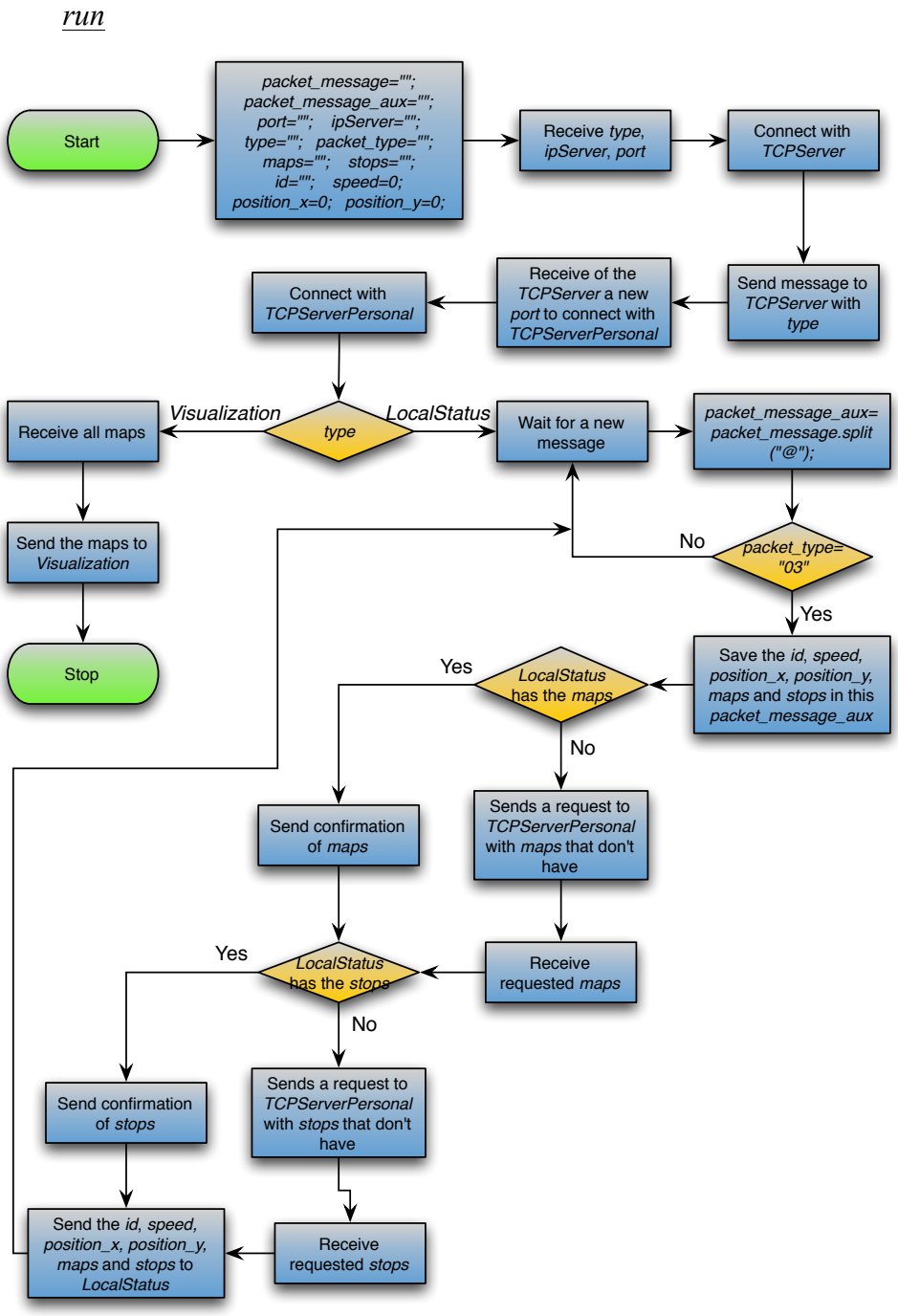

Figura 4.25 - Fluxograma elucidativo do funcionamento da função *run*

Esta função `run` começa por estabelecer uma conexão com o *TCPServer* que posteriormente lhe informa a porta que ele terá que pedir nova conexão desta vez com o *TCPServerPersonal*. Uma vez estabelecida a comunicação estável, a classe entra numa execução cíclica que varia em função do seu tipo de funcionamento, ou seja, difere o funcionamento em função de onde está inserido, no *LocalCoordinator* ou na *Visualization*. É perceptível

pelo fluxograma apresentado acima, que de cada vez que é recebido uma mensagem do tipo `"03"`, para a criação de um novo *Actor*, é feita a verificação da existência ou não dos mapas especificados nas variáveis `maps` e `stops`. Caso esses mapas ainda não existam no *LocalCoordinator* a que foi atribuído o nó, têm de ser pedidos e recebidos por este mesmo interface, para que a troca de informação seja confirmada. No caso de o *LocalCoordinator* já possuir os mapas necessários apenas tem de enviar uma confirmação ao servidor. Na Figura 4.26 é mostrado um exemplo da criação de um ator em que o *LocalStatus* já possui os mapas.

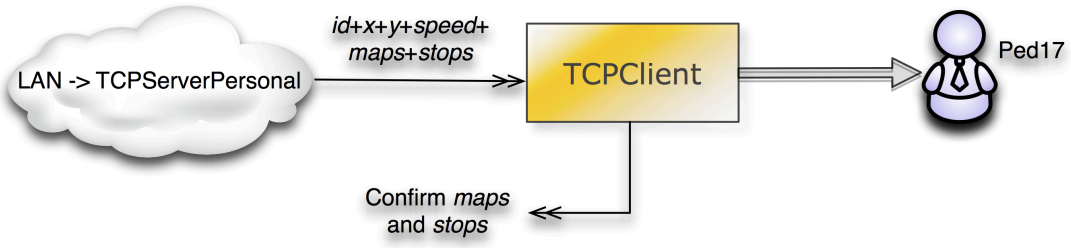

Figura 4.26 - Exemplo de criação de um novo *Actor* em que o *LocalStatus* já possui os mapas

4.4.2.3 TCPServer

Aquando da exposição do BartUM Simulator, foi apresentada a entidade *TCPServer* que é um servidor TCP que foi implementado apenas para receber novas conexões TCP que pretendam ser efetuadas.

Para que este servidor esteja sempre operacional a receber novas ligações, que podem surgir de novos *LocalCoordinators* ou *Visualization*, teve de ser implementado como sendo um processo individual, ou seja, uma `Thread`. De forma a que o `TCPServer` esteja sempre operacional a receber novas ligações, cada vez que ele recebe uma conexão limita-se a enviar as informações necessárias para que se ligue a um *TCPServerPersonal* que ele irá criar apenas para tratar daquela entidade. Esta entidade apenas está presente no *GlobalCoordinator* (Figura 4.1), pois é ele quem serve os *LocalCoordinators* ou a *Visualization* com os mapas e os novos nós da simulação.

4.4.2.3.1 Variáveis Globais

Ao nível das variáveis globais, o `TCPServer` possui seis, sendo algumas delas semelhantes aos que também foram implementadas no `TCPClient`.

A variável `port` é, tal como no cliente TCP, a porta que estará sobe escuta, neste caso, pelo servidor. Assim, qualquer pedido de conexão que se pretenda fazer tem de ser feito a esta porta para que o servidor possa dar o tratamento devido ao pedido. O valor de `port`, obrigatoriamente inteiro, é passado à classe no seu construtor, sendo essa a única variável que é iniciada nesse método. Para que todos os clientes saibam em que porta se encontra o servidor, esta tem de ser fixa e conhecida por todas as entidades (ficheiro de configuração).

Tal como acontece em todas as classes do *package communication* já apresentadas, tiveram de ser implementadas variáveis relacionadas com as mensagens. Nesta classe foram implementadas três variáveis relacionadas com as mensagens: o `packet_message`, que, tal como no `TCPClient`, é a mensagem que é enviado ou recebido pela classe; o `packet_type`, que alberga o tipo de mensagem que chega ou que é enviado; o `packet_message_aux` é uma variável que é usada na divisão da mensagem que chega à classe, `packet_message`, nos seus vários campos. A divisão que é feita é a mesma que a apresentada nas classes anteriores, ou seja, os diferentes campos são divididos pelo caractere "@".

Como esta variável tem um tratamento diferente em função do tipo de cliente que se liga, tem de existir uma variável, `client`, onde é guardado o tipo de cliente que fez o pedido de conexão, ou seja, só pode ter dois tipos de informação lá guardados, ou `"localStatus"` ou `"visualization"`;

Uma vez que quem faz a gestão das ligações TCP que vão sendo feitas ao logo da simulação é esta classe, `TCPServer`, existe a necessidade de haver um controlo sobre as portas que estão a ser usadas de forma a não existir nenhuma sobreposição. Para que isso fosse feito, foi implementada a variável `portUsed` que não é mais que uma lista com todas as portas que estão a ser usadas para comunicação TCP. Assim, quando uma nova ligação for aceite, tem que se verificar se a porta está ou não a ser usada antes de ser atribuída aquela nova ligação. Como a ligação feita com a *Visualization* é momentânea, ou seja, dura apenas o tempo necessário para serem enviados todos os mapas, a porta que é usada para essa conexão nunca é incluída nesta lista. Pela mesma razão, a ligação feita com essa entidade é feita através de uma porta fora do intervalo que é usada pelos *LocalCoordinator*, garantindo assim que não há uma sobreposição, nem que seja momentânea.

4.4.2.3.2 Funções

Mais uma vez, a classe `TCPServer` apenas possui uma função que, uma vez que se trata de uma classe que tem um funcionamento independente, se trata da função `run`. Esta função, descrita pelo fluxograma presente na Figura 4.27, está constantemente a atender pedidos e a reencaminha-los para servidores dedicados apenas aquele cliente.

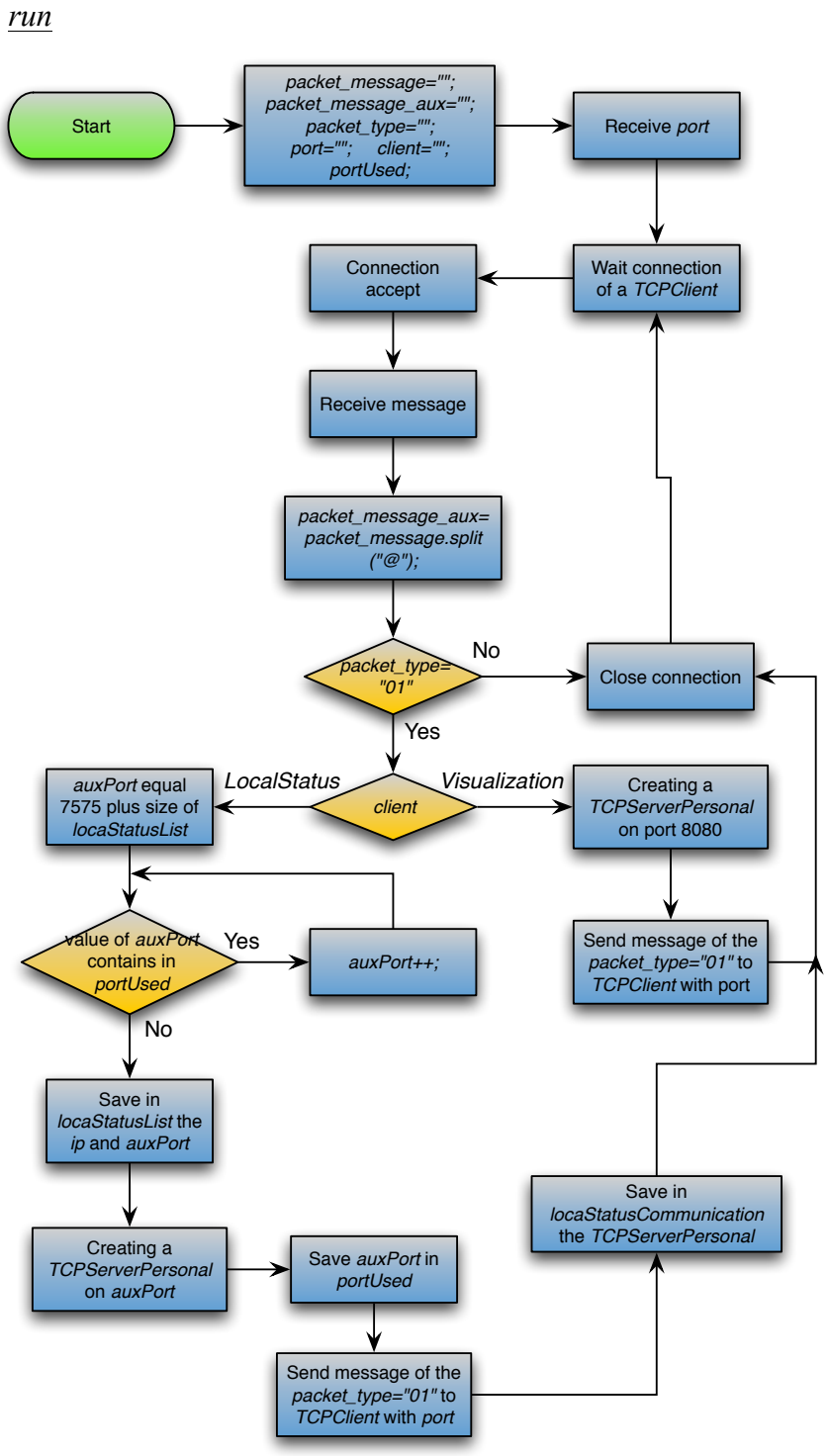

Figura 4.27 - Fluxograma da execução da função *run*

É nesta função que tudo o que é feito pela classe *TCPServer* é implementado, sendo a garantia de que é feito ciclicamente a colocação de um ciclo logo no início da sua implementação que fica à espera da chegada de novos pedidos de conexão. Como foi dito, esta

função está sempre à espera que novas conexões TCP sejam pedidas ao *GlobalCoordinator*, sendo que essa espera se mantém durante todo o tempo da simulação. Isso permite que, durante uma simulação, se o utilizador achar necessário que seja acrescentado um novo *LocalCoordinator* é possível, bastando para isso que seja ativado e faça um pedido de ligação TCP. Outro factor que faz com que seja necessário que este servidor esteja sempre ativo é o facto de uma *Visualization* ser inserida na simulação, e isso pode acontecer a qualquer momento da mesma. Essa conexão tem que se dar a conhecer através deste servidor, porque é via TCP que a *Visualization* receberá os mapas, enviados pelo *GlobalCoordinator*.

Uma vez recebido um pedido de conexão, esta classe tem duas forma de atuar que são diferenciadas pelo cliente que fez o pedido. Uma vez que este servidor se encontra no *GlobalCoordinator*, ele apenas recebe pedidos de conexão por parte de *LocalCoordinator* ou *Visualization*. Estes dois tipos de cliente tem que ser diferenciados porque o seu tipo de conexão é diferente, pois as conexões com o *LocalCoordinator* serão permanentes, durante a simulação, e as conexões com a *Visualization* são temporárias, demorando apenas o tempo necessário para o envio dos mapas. Contudo, ambos os clientes levam à criação de um *TCPServerPersonal*. No entanto, como apenas as ligações com o *LocalCoordinator* são duradouras, as ligações feitas com a *Visualization* são efetuadas através de uma única porta pré-definida, 8765, e que não entra na lista `portUsed`.

4.4.2.4 TCPServerPersonal

Para terminar a descrição das classes do *package communications*, falta abordar a implementação da classe *TCPServerPersonal*. Esta classe é também um servidor TCP diferindo do apresentado anteriormente no facto de ser dedicado apenas a uma conexão, ou seja, a um só cliente. Tal como acontece com o *TCPServer*, esta classe está alojada do lado do *GlobalCoordinator* (Figura 4.1).

Pelas mesmas razões que foram apresentadas para as restantes classes deste *package*, também a classe `TCPServerPersonal` teve de ser implementada como sendo uma `Thread`.

4.4.2.4.1 Variáveis Globais

No caso da classe `TCPServerPersonal`, as variáveis globais não diferem muito daqueles que já foram apresentadas nas classes pertencentes ao protocolo TCP.

A variável `port` é, como nas restantes classes já apresentadas, o valor numérico que indica a porta pela qual está a ser feita a comunicação. Já no caso da variável `newActor` é a primeira vez que surge neste contexto. Esta variável funciona como uma *label* para o envio de novos atores ao seu cliente TCP. Esta variável é necessária pois esta classe, ao contrário do cliente, apenas envia informação, mas não tendo mecanismo para saber quando enviar essa informação torna-se difícil saber quando o fazer. Para colmatar este problema, foi criada esta variável que está sempre a 0, tal como é inicializada, e que quando o *GlobalCoordinator* pretende enviar informação para criar um novo *Actor* através desta interface a coloca a 1.

Nesta classe foram implementadas 7 variáveis que são usadas para guardar informação tanto da sua ligação como informações temporárias relativas ao nó a ser criado. As variáveis `ip_client`, `type`, `packet_type` e `packet` são relativas à ligação, sendo o `ip_client` o IP do seu cliente TCP, o `type` o tipo de cliente, podendo apenas ser `"localStatus"` ou `"visualization"`, o `packet_type` o tipo de mensagem que foi recebido ou vai ser enviado, e o `packet` mensagem recebida ou a enviar. Uma vez que as mensagens são trocadas como sendo objetos há a necessidade de se decompor as mensagens que vão chegando pelos diversos campos. Tal como acontece com as outras classe do *communications*, teve de ser criada uma variável, para que seja possível fazer essa divisão e que os campos sejam todos de fácil acesso. Nesta classe, a variável que possui a mensagem recebida decomposta pelos seus diversos campos é denominada de `packet_aux`.

Como são estas classes que vão enviar para os *LocalCoordinators* a informação relativa aos novos nós a serem criados, tem que ter essa informação guardada antes de a enviar. As variáveis relativas a esses novos atores são `id`, `position_x`, `position_y`, `speed`, `maps` e `stops`, sendo o `id` o *id* do novo *Actor* a ser enviado, as `position_x` e `position_y` os valores da posição inicial do ator, a `speed` velocidade inicial, o `maps` os nomes de todos os mapas necessários ao *Actor* e o `stops` todos os locais de paragem relativos ao *Actor* a ser enviado.

4.4.2.4.2 Funções

Sendo a classe `TCPServerPersonal` implementada para funcionar como um processo diferente, ou seja, como uma *thread*, teve que ser implementado a função `run` para funcionar como tal. Uma vez que podem existem dois tipos de *TCPClient*, também o *TCPSer-*

verPersonal tem que ter dois modos de executar, como é perceptível pelo fluxograma da Figura 4.28.

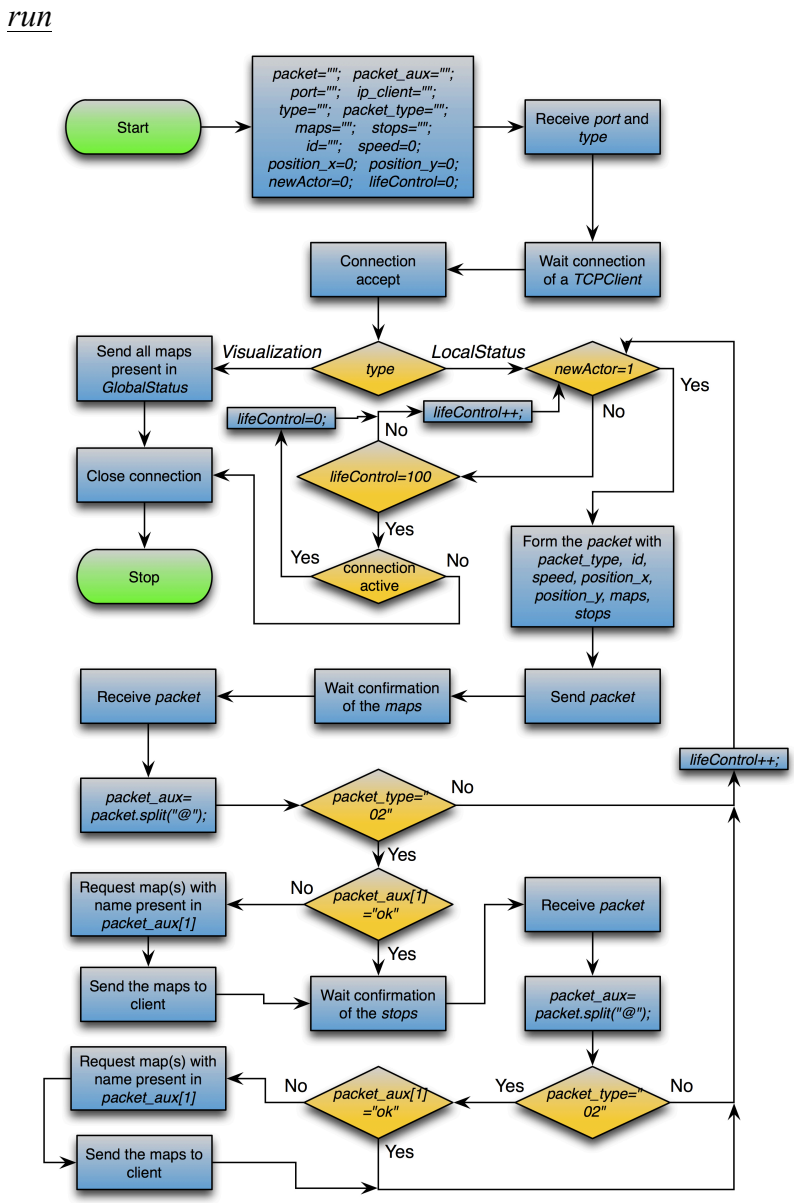

Figura 4.28 - Fluxograma da execução da função *run*

Uma vez que a informação relativa à porta e ao tipo de cliente é fornecida durante a construção da classe, esta método apenas tem que esperar que o cliente TPC se ligue a si para começar a trocar informação logo que necessário. Como se tinha como objetivo que as classes fossem o mais possível reutilizáveis, ou seja, poder-se usar a mesma classe para os diferente tipo de cliente, foi necessário criar dois modos diferente da função `run` trabalhar.

O primeiro modo de funcionamento é no caso de o cliente TCP ser de um *LocalCoordinator*. Nesse caso, o método `run` entra num ciclo de teste infinitamente verdadeiro, e sempre que a *label* `newActor` for ativa, ou seja, estiver a 1, este tem de fazer uma mensagem com toda a informação relativa ao *Actor*, que lhe foi colocado nas respectivas variáveis, e enviar tudo numa mensagem para o seu cliente. De cada vez que é enviado um novo *Actor* para o *LocalCoordinator* a função fica à espera de informação relativamente aos mapas necessários, seja a confirmação positiva ou negativa. Enquanto o valor do `newActor` estiver em 0, o ciclo vai fazendo sucessivamente o teste a esta variável. Para haver um controlo de existência da conexão, a cada 100 ciclos efetuados que não seja enviado qualquer *Actor*, esta função faz um teste para verificar se a ligação ainda está ou não ativa. Caso não esteja, o *socket* é fechado e é aberto outro semelhante para manter as comunicações ativas.

O segundo modo de funcionamento, é quando o seu *TCPClient* é a *Visualization*. Neste acaso, o `TCPServerPersonal` limita-se a enviar ao seu cliente todos os mapas presentes no *GlobalStatus* e a fechar a conexão. Isto acontece porque este tipo de cliente não tem necessidade de fazer mais nenhum tipo de interação com *GlobalCoordinator* durante a simulação.

4.5 Actor

O *package um.simulator.actor* é, como já foi mencionado anteriormente, onde são implementadas as classes referentes aos diferentes atores. A implementação das classes referentes aos *Actors* não está abrangida nos objetivos desta dissertação, contudo, para a minimização da informação guardada e transmitida, foi criada uma classe usada apenas para guardar as informações relevantes de cada ator. Como os desafios de optimização da informação estão incluídos nos objetivos desta dissertação, a classe implementada para esse fim, *ActorStatus*, é aqui contemplada.

4.5.1 ActorStatus

Esta classe é muito diferente das outras aqui apresentadas pois não tem qualquer funcionalidade a não ser a de guardar a informação relevante de cada *Actor*. Para isso, esta classe permite que outras classes possam manusear a informação colocada nela.

4.5.1.1 Variáveis Globais

Embora as outras classes possam ter acesso à informação guardada nesta, não é possível que uma outra classe consiga alterar a informação aqui guardada. Assim, uma vez que seja atribuído um valor às variáveis, só a própria classe os pode alterar, no entanto isso nunca acontecerá.

Uma vez que esta classe é usada para albergar a informação relevante de cada nó, necessita de ter variáveis semelhantes às existentes num ator. Assim, teve de ser implementada uma variável para guardar o *id* do *Actor*, `actor_id`.

Outra informação muito relevante de um ator é a sua posição atual, pelo que tiveram de ser implementadas variáveis para se guardarem as coordenadas *x* e *y*, a `actor_x` e a `actor_y`, respectivamente.

Para que a informação guardada neste objeto tenha uma referencia temporal, de forma a controlar se é recente ou não a informação que se esta a aceder, teve de ser criada uma variável que não existe nos *Actor* que é a `actor_time`. Esta variável vai ser composta por uma informação temporal do sistema. Assim, quando é criado o objeto, é pedido ao sistema uma referência temporal que é guardada nesta variável. É com base neste tempo que serão detectados *Actors* que possivelmente estão desativados.

Apesar de neste momento da implementação apenas se considerarem estas variáveis como sendo as imprescindíveis para caracterizar um *Actor*, a implementação desta classe foi feita de forma a que no futuro se possa guardar mais informação que se ache relevante. Para isso é necessário apenas acrescentar-se a variável e atualizar o construtor.

4.5.1.2 Funções

A classe `ActorStatus` não apresenta qualquer função, pois este objeto é completamente estático. Assim, apenas recebe a informação de um ator e guarda nas respectivas variáveis.

Sempre que um *Actor* efetua um movimento tem de o comunicar ao *LocalStatus* a que se encontra associado para que este mantenha a sua informação atual. Esta comunicação dos *Actor* com o *LocalStatus* é feita através desta classe, ou seja, os *Actor* sempre que se movimentam criam um novo objeto deste tipo onde guardam a sua nova posição e enviam esse mesmo objeto ao *LocalStatus* que o substitui na sua lista de atores, `actorList`. A partir desse momento, esta classe passa a ser a representação do *Actor* nos *LocalStatus*, no

GlobalStatus e na *Visualization*. Na Figura 4.29 é demonstrado um exemplo de um *Actor* a guardar uma movimentação sua na tabela dos atores, `actorList`.

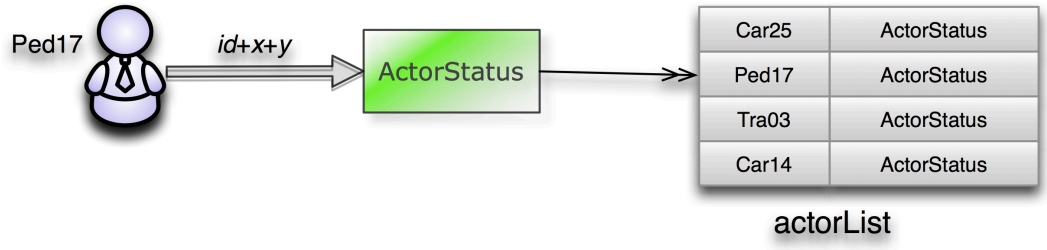

Figura 4.29 - Exemplo de uma atualização do nó *Ped17*

Como a `actorList` foi implementada como sendo uma tabela em que a chave é o *id* de cada nó, apenas tem que inserir o novo `ActorStatus` na lista com o seu *id* para que seja efectuada a substituição do antigo `ActorStatus` pelo novo.

5 Testes e Análise de resultados

Durante este capítulo, serão apresentados os testes que foram efetuados de forma a garantir o correto funcionamento do núcleo do simulador desenvolvido, garantindo o cumprimento dos objetivos inicialmente propostos.

Desta forma, foram efetuados 6 tipos de testes, com o intuito de colocar o BartUM Simulator no seu máximo para garantir a sua estabilidade. Os testes realizados foram os seguintes: testes de funcionalidade, testes de distribuição de carga, testes de estabilidade, testes de carga dos diversos processadores, testes do momento em que a carga passa a ser crítica e também um teste a nível informativo a um simulador semelhante que foi estudado, o The ONE.

Uma vez terminados os testes, é feita uma análise comparativa do que eram os resultados esperados e aqueles que se obtiveram, sendo feita também uma interpretação desses mesmos resultados no contexto do simulador desenvolvido.

Para a realização dos testes foram utilizados 4 computadores diferentes, logo com componentes também eles diferentes, apresentados na Tabela 5.1.

	Computador 1	**Computador 2**	**Computador 3**	**Computador 4**
Sistema Operativo	Windows 7 Home Premiun (64 bits)	Windows 7 Enterprise (32 bits)	Windows 7 Professional (32 bits)	Mac OS X versão 10.6.8 (32 bits)
Processador	Intel(R) Core(TM) i7 CPU Q740 @1,73 GHz 1,73 GHz	Intel(R) Core(TM) 2 Duo CPU T7500 @2,20 GHz 2,20 GHz	Genuine Intel(R) CPU T2130 @1,86 GHz 1,86 GHz	Intel Core 2 Duo @2,20 GHz 2,20 GHz
Memoria RAM	8 GB	2GB	1GB (756 utilizável)	1GB

Tabela 5.1 - Características principais dos computadores utilizados nos testes

Na execução dos testes, embora apareçam 3 tipos de atores, *Tra*, *Ped* e *Car*, todos eles tem um comportamento completamente aleatório, ou seja, nenhum deles tem para já

um tipo de movimento específico. Assim, o que difere entre eles é o ritmo a que são gerados pelos respetivos *Generators*.

5.1 Funcionalidade

O teste de funcionalidade consiste em testar as funções que foram implementadas no simulador de forma a saber se têm o comportamento esperado. Neste teste será testado o funcionamento do sistema em geral, ou seja, o funcionamento do *GlobalCoordinator* e dos *LocalCoordinators*. Assim, serão feitos testes de comunicação e a respectiva interpretação da informação trocada pelas duas entidades, demonstrando-se assim, que o núcleo do Bar-tUM Simulator tem um funcionamento correto e segundo o esperado.

Na realização deste teste foi usado o computador 4 como *GlobalCoordinator* e os restantes 3 como *LocalCoordinator*.

5.1.1 Resultados

As figuras seguintes ilustram o comportamento dos 4 computadores.

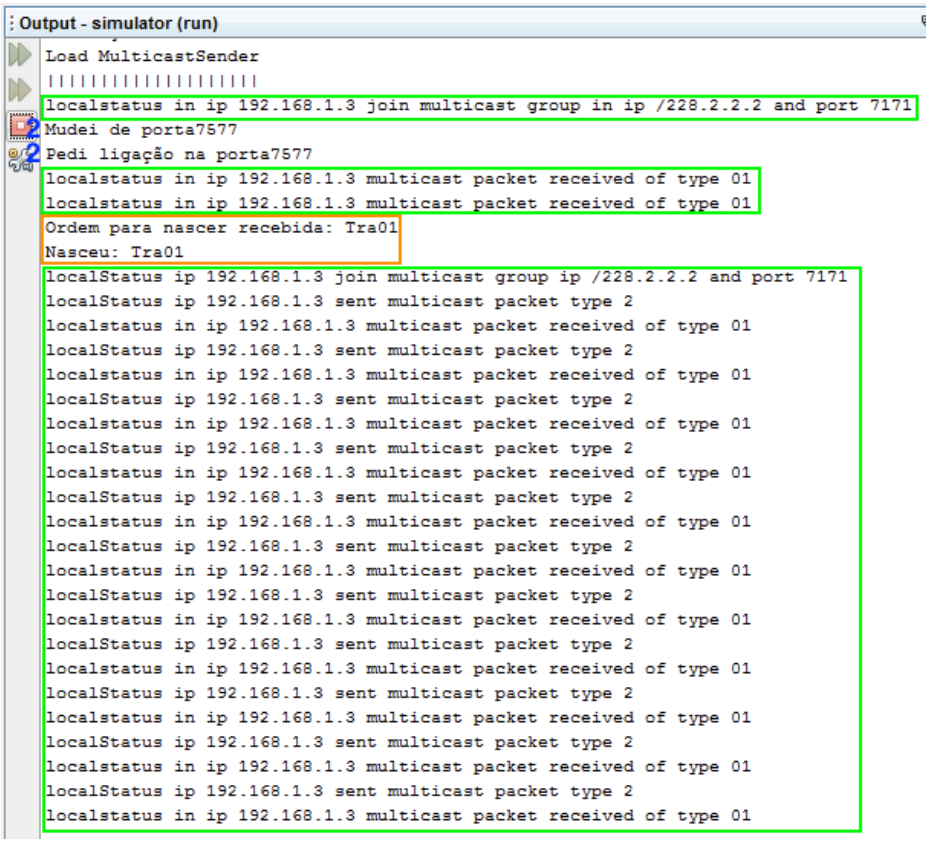

Figura 5.1 – Lista de eventos observados no computador 1 (*LocalCoordinator*)

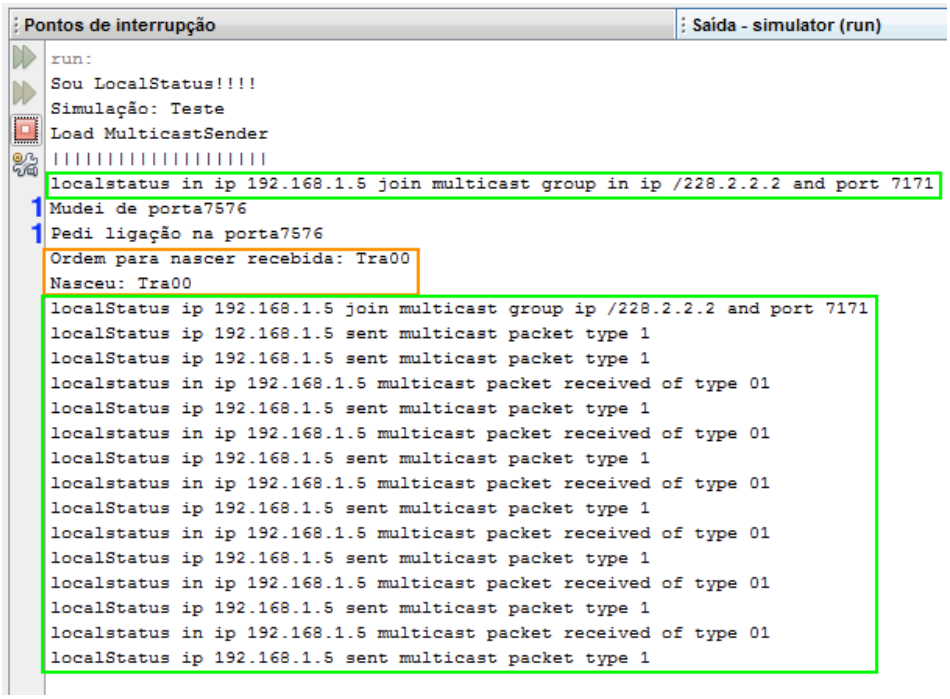

Figura 5.2 - Lista de eventos observados no computador 2 (*LocalCoordinator*)

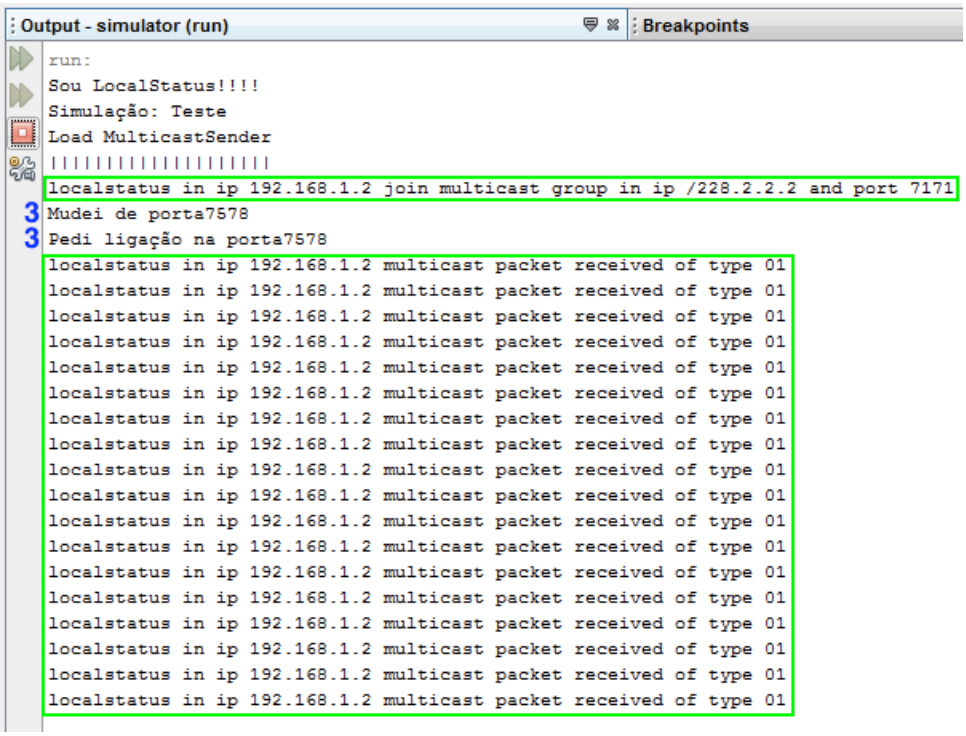

Figura 5.3 - Lista de eventos observados no computador 3 (*LocalCoordinator*)

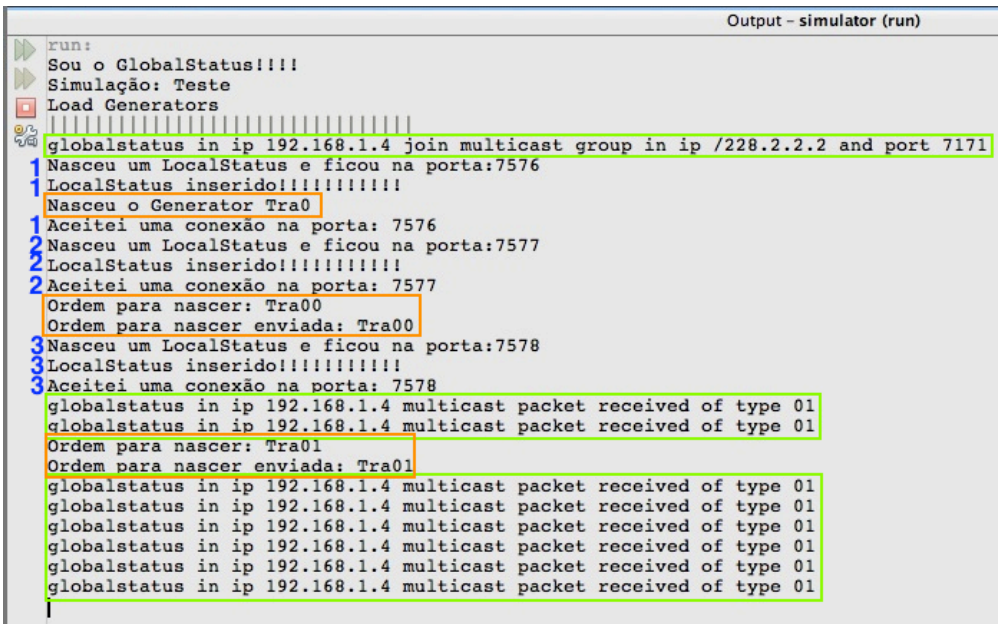

Figura 5.4 - Lista de eventos observados no computador 4 (*GlobalCoordinator*)

5.1.2 Análise de resultados

Com a concretização deste teste esperava-se observar resultados ao nível da comunicação *multicast*, da comunicação TCP, da criação de geradores e da criação de atores.

5.1.2.1 Comunicação multicast

Segundo o que foi apresentado no capítulo da implementação, o envio de mensagens *multicast* só começa a ser efetuado pelos *LocalCoordinator* quando estes têm *Actors* associados a si. No entanto, a interface de escuta do grupo *multicast* tem que estar ativada logo que o *LocalCoordinator* é ligado, de forma a ir recebendo informações sobre o estado da simulação enquanto aguarda que atores lhe sejam atribuídos.

Como se pode ver pelos resultados apresentados em 5.1.1, o resultado foi o esperado. Em todas as figuras apresentadas, a primeira informação relevante que surge é o pedido de *join* ao grupo *multicast* (primeiro retângulo verde) por parte da interface de entrada (*MulticastReceiver*).

Uma vez que o computador 2 é o primeiro a receber ordem para criar um *Actor* (Tra00), é quem primeiro faz o *join* da interface de saída (*MulticastSender*) e começa a enviar mensagens *multicast* para o grupo, como se pode ver na Figura 5.2. Os restantes computadores começam, nesse momento, a receber as mensagens que são enviados para o grupo. O próximo computador que recebe ordem para a criação de um actor é o computador 1,

que segue o mesmo processo que o anterior, começando assim a enviar mensagens para a rede (Figura 5.1).

Já o computador 3, como não chegou a ter nenhum ator associado a si não ativou a interface de saída *multicast*, pelo que até àquele momento apenas recebeu as mensagens que foram enviados para a rede pelos restantes *LocalCoordinators*, como é perceptível na Figura 5.3.

O computador 4, representado na Figura 5.4, é o computador destinado a ser o *GlobalCoordinator*, como tal apenas recebe as mensagens que foram enviadas para a rede. Como é perceptível pela Figura 5.4, este só começa a receber informação do grupo *multicast* depois de ter dado ordem para a criação de pelo menos um *Actor*, tal como era esperado.

Considerando a importância existente na rápida difusão deste tipo de mensagens mas também a importância de as mesmas chegarem ao seu destino, é de referir que todas as mensagens chegaram em tempo útil aos seus destinos.

5.1.2.2 Comunicação TCP

Nos resultados apresentados em 5.1.1 é também possível comprovar o funcionamento das conexões TCP. Aquando da especificação destas conexões foi descrito que iriam ser efetuadas apenas para a comunicação entre o *GlobalCoordinator* e os *LocalCoordinators*. Assim, cada *LocalCoordinator* começa por pedir uma conexão TCP a uma porta bem conhecida, sendo posteriormente atribuída uma outra porta para que não haja colisões.

Esta troca de informação é observada nas figuras apresentadas em 5.1.1, sendo a correspondência feita pelo números 1, 2 e 3 para mostrar a conexão dos 3 *LocalCoordinators* ao *GlobalCoordinator*. Desta forma, na Figura 5.4 podem ver-se os pedidos de conexão ao *GlobalCoordinator*, sendo que o primeiro pedido, representado pelo número 1 nas figuras, é feito pelo computador 2 (Figura 5.2). O segundo computador a fazer a conexão TCP é o computador 1 (Figura 5.1) e a comunicação é identificada como 2 nas figuras. Por último, o computador 3 (Figura 5.3) faz também ele o pedido de conexão TCP ao *GlobalCoordinator*, sendo esta ligação identificada com o número 3.

Uma vez efetuadas as ligações TCP, a forma de garantir que estão em funcionamento é a troca de informação entre o *GlobalCoordinator* e os *LocalCoordinators* de forma a serem criados novos atores para a simulação. Isso pode ser observado nos retângulos laranja

presentes na Figura 5.1, Figura 5.2 e Figura 5.4 onde são identificados os *Actors* que já foram criados.

5.1.2.3 Criação de Generators

Na descrição do funcionamentos do simulador explicou-se que os *Generators* só seriam iniciados quando o *GlobalCoordinator* recebesse pelo menos uma conexão TCP com um *LocalCoordinator*. De forma a ser mais fácil controlar o número de atores pretendido neste teste apenas foi criado um *Generator*.

Como se pode ver na Figura 5.4, referente ao computador com o *GlobalCoordinator*, os resultados foram tal e qual os esperados, ou seja, só após a conexão apresentada com o número 1 é que é criado o *Generator* (primeiro retângulo laranja). A partir do momento em que o *Generator* é criado começam também a ser criados e enviados para os *LocalCoordinators* os novos atores.

5.1.2.4 Criação de Actors

Por fim, através deste teste é possível observar se a criação e distribuição dos *Actors* está ou não a ser feita. O que se esperava era que o *GlobalCoordinator*, através do *Generator*, criasse novos atores e os enviasse ao *LocalCoordinator*. Por sua vez, o *LocalCoordinator* tem de ser capaz de receber essa ordem de criação e criar o ator.

Também esta funcionalidade obteve os resultados esperados, ou seja, os *Actors* que vão sendo criados pelo *GlobalCoordinator* são enviados para os *LocalCoordinators* e estes conseguem criar esses mesmos atores.

Nas Figura 5.4 pode ser observada a criação de 2 *Actors*, o Tra 00 e o Tra 01, que são enviados ao computador 2 (Figura 5.2) e ao computador 1 (Figura 5.1) respectivamente.

5.2 Distribuição de carga

Uma vez que o sistema desenvolvido é distribuído por diversas máquinas, foi implementada uma função no *GlobalStatus* que efetua a distribuição da carga. Assim, este teste consiste em verificar se essa distribuição está a ser executada corretamente.

Das duas variáveis descritas para esta distribuição, foi usada a referente ao número de nós ativos. Embora seja o método menos eficaz, quando comparado com a carga do processador, foi o utilizado devido ao facto de, nas versões mais recentes do *Windows*, não es-

tar disponível no Java a resposta ao pedido de carga do processador. Isto acontece porque na sua implementação, esta função foi considerada demasiado dispendiosa para o sistema operativo.

Para atestar essa correta distribuição, em qualquer momento temporal, a diferença entre o número total de atores associados aos vários *LocalCoordinators*, se forem iniciados ao mesmo tempo, não pode ser superior a 3. A meta dos 3 atores de diferença foi o considerado de referência tendo em consideração a implementação feita, uma vez que a diferença entre o número de atores que é enviado do *LocalCoordinator* para o *GlobalCoordinator* e o número de atores que efetivamente possui não deve ser maior que 2. A diferença pode chegar a 2 no caso de um pacote de atualização se perder ou então no caso de o débito de criação de atores estar a ser muito maior que o débito da atualizações do *LocalCoordinator*.

Tal como no teste anterior, ao computador 4 foi atribuída a função de *GlobalCoordinator*. Assim, apenas interessa observar o comportamento dos computadores 1, 2 e 3.

5.2.1 Resultados

As figuras seguintes ilustram o comportamento dos computadores 1, 2 e 3.

```
Sou LocalCoordinator!!!!
SimulaÞòo: Teste
Load MulticastSender
!!!!!!!!!!!!!!!!!!!
O meu IP e: 192.168.1.2
Nasceu: Ped00
Tenho 1.0 atores associados, de um totsl de 3 atores!
Tenho 1.0 atores associados, de um totsl de 3 atores!
Tenho 1.0 atores associados, de um totsl de 4 atores!
Tenho 1.0 atores associados, de um totsl de 4 atores!
Tenho 1.0 atores associados, de um totsl de 4 atores!
Tenho 1.0 atores associados, de um totsl de 4 atores!
Tenho 1.0 atores associados, de um totsl de 4 atores!
Tenho 1.0 atores associados, de um totsl de 4 atores!
Tenho 1.0 atores associados, de um totsl de 5 atores!
Tenho 1.0 atores associados, de um totsl de 5 atores!
Tenho 1.0 atores associados, de um totsl de 5 atores!
Tenho 1.0 atores associados, de um totsl de 5 atores!
Tenho 1.0 atores associados, de um totsl de 5 atores!
Tenho 1.0 atores associados, de um totsl de 5 atores!
Tenho 1.0 atores associados, de um totsl de 5 atores!
Nasceu: Ped02
Tenho 2.0 atores associados, de um totsl de 6 atores!
Tenho 2.0 atores associados, de um totsl de 6 atores!
Tenho 2.0 atores associados, de um totsl de 6 atores!
Tenho 2.0 atores associados, de um totsl de 6 atores!
Tenho 2.0 atores associados, de um totsl de 6 atores!
Tenho 2.0 atores associados, de um totsl de 7 atores!
Tenho 2.0 atores associados, de um totsl de 7 atores!
Tenho 2.0 atores associados, de um totsl de 7 atores!
Tenho 2.0 atores associados, de um totsl de 9 atores!
Tenho 2.0 atores associados, de um totsl de 10 atores!
Nasceu: Ped05
Tenho 3.0 atores associados, de um totsl de 12 atores!
Tenho 3.0 atores associados, de um totsl de 12 atores!
Tenho 3.0 atores associados, de um totsl de 12 atores!
Tenho 3.0 atores associados, de um totsl de 12 atores!
```

Figura 5.5 – Comportamento do computador 1

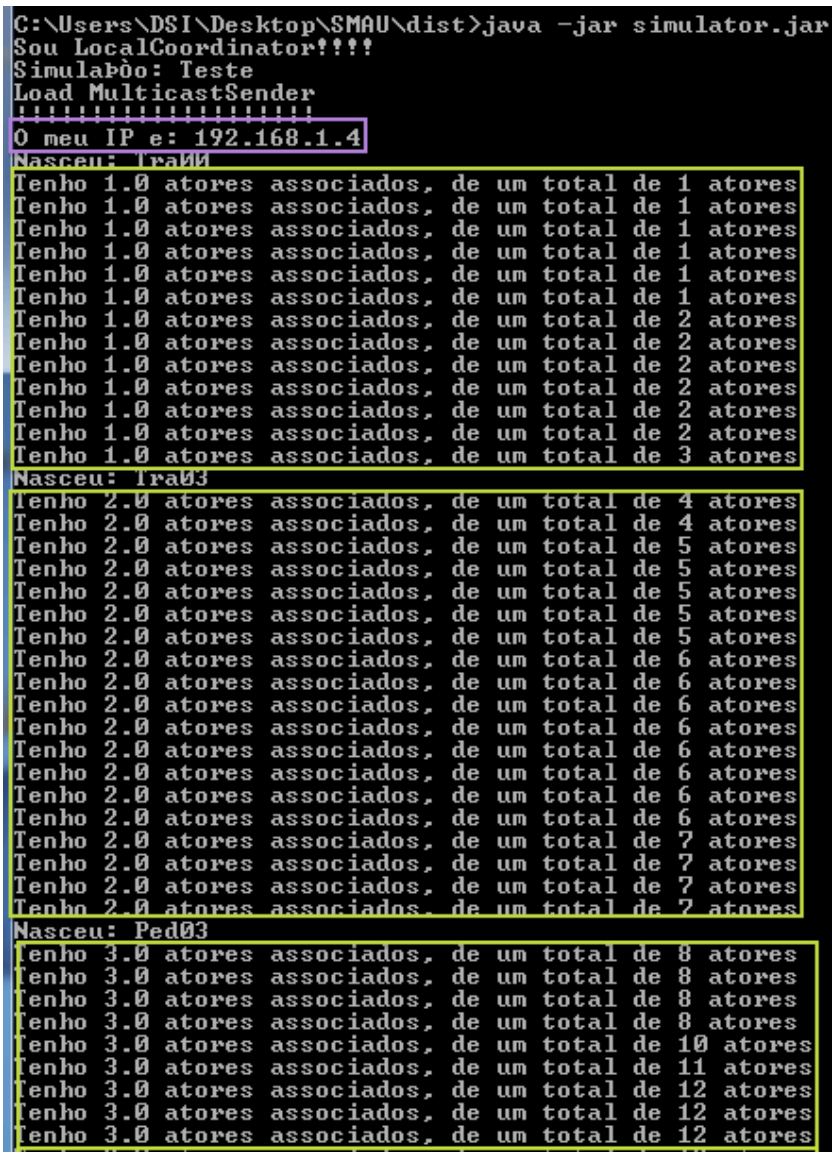

Figura 5.6 - Comportamento do computador 2

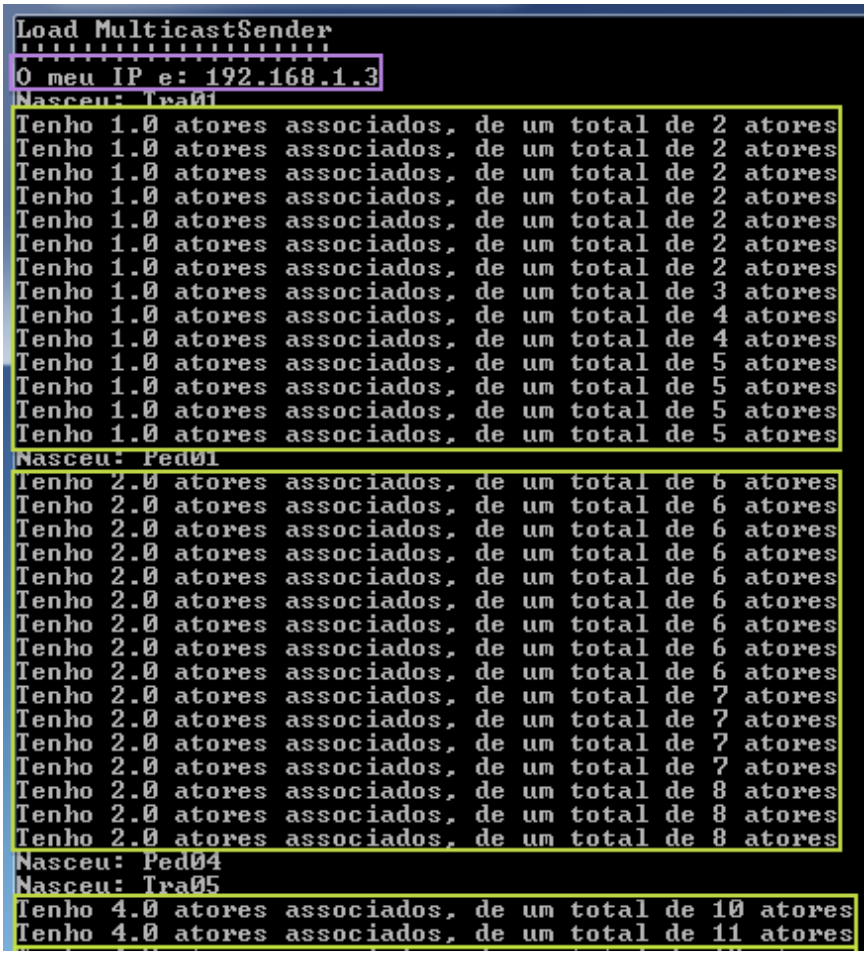

Figura 5.7 - Comportamento do computador 3

5.2.2 Análise de resultados

Para a análise dos resultados deste teste, teve-se como ponto de partida o facto de a opção para a distribuição dos atores, pelos diversos *LocalCoordinator*, ser feita pelo número de *Actors* que cada um tem já associado. Assim, o número de atores em cada um dos *LocalCoordinator* tem que ser semelhante em todos eles. Como se pode ver pelas figuras apresentadas acima, os resultados obtidos no teste de distribuição de carga foram os que se esperava.

Na realização deste teste, os 3 computadores que albergam o *LocalCoordinator* foram iniciados aproximadamente ao mesmo tempo. A distribuição vai sendo efetuada à medida que o *GlobalStatus* vai dando ordens para criar novos atores. Pelas mensagens presentes nas Figura 5.5, Figura 5.6 e Figura 5.7, os 3 *LocalCoordinator* presentes neste teste vão tendo um número de atores associados sempre muito próximo, fazendo com que este teste tenha sido bem sucedido.

5.3 Estabilidade

Um dos requisitos do simulador é ser multiplataforma e não ser demasiado exigente em termos de *hardware*. Outra consideração a ter em conta é que o simulador não irá funcionar sobre um computador dedicado apenas aquela função, ou seja, para medir a sua estabilidade tem que ser considerado também o funcionamento do respetivo sistema operativo em que está inserido.

Assim, este teste consiste em colocar diversos computadores com diferente componentes e sistemas operativos a funcionar como *LocalCoordinator* e verificar qual o comportamento do computador ao longo do tempo de uma simulação.

A concretização deste teste vai permitir que seja verificada a estabilidade do funcionamento em diversas plataformas e também que seja encontrado o ponto em que o sistema funciona de forma mais estável.

Contudo, para que fosse testada a estabilidade de todos os computadores quando albergavam o *LocalCoordinator*, este teste foi dividido em duas fase, uma em que o computado 4 é o *GlobalCoordinator* e uma outra em que passa a ser o computador 1 que alberga o *GlobalCoordinator*.

Num primeira fase de teste manteve-se o panorama habitual, ou seja, o computador 4 como *GlobalCoordinator* e os restantes 3 como *LocalCoordinators*. Por forma a ser medida a estabilidade dos computadores foi anotado o desemprenho, em percentagem, da atividade dos processadores e da ocupação da memória RAM.

5.3.1 Resultados

Para que seja possível comparar o estado de estabilidade do computador antes e depois do funcionamento do simulador, foi necessário fazer um levantamento do estado inicial de todos os computadores.

	Computador 1	Computador 2	Computador 3	Computador 4
Processador	3%	2%	3%	2%
Memória RAM	26%	57%	47%	59%

Tabela 5.2 - Estado inicial dos computadores utilizados nos testes efetuados

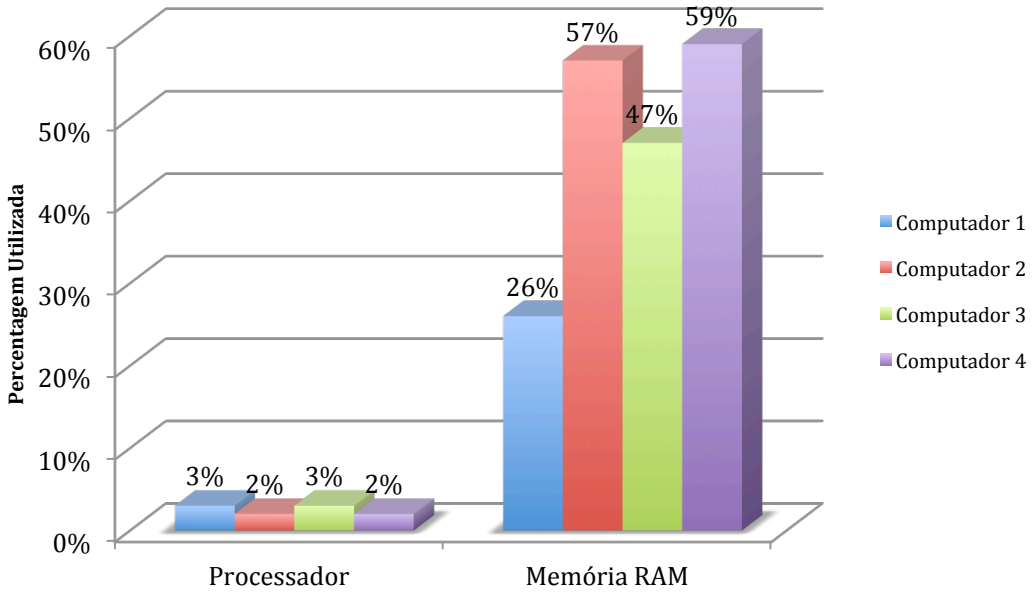

Figura 5.8 - Estado inicial dos computadores

Nº de atores (por *Local-Coordina-tor*)	Computador 1		Computador 2		Computador 3		Computador 4	
	Processa-dor	Memória RAM	Processador	Memória RAM	Processador	Memória RAM	Processador	Memória RAM
0	3%	26%	2%	57%	3%	47%	2%	59%
10	1%	31%	2%	50%	5%	51%	100%	66%
20	1%	31%	2%	50%	6%	54%	99%	66%
30	1%	31%	2%	50%	6%	54%	99%	66%
40	1%	31%	2%	50%	6%	53%	99%	66%
50	1%	31%	2%	50%	6%	53%	99%	66%

Tabela 5.3 - Percentagem de utilização de processador e memória RAM em função dos nº de nós ativos

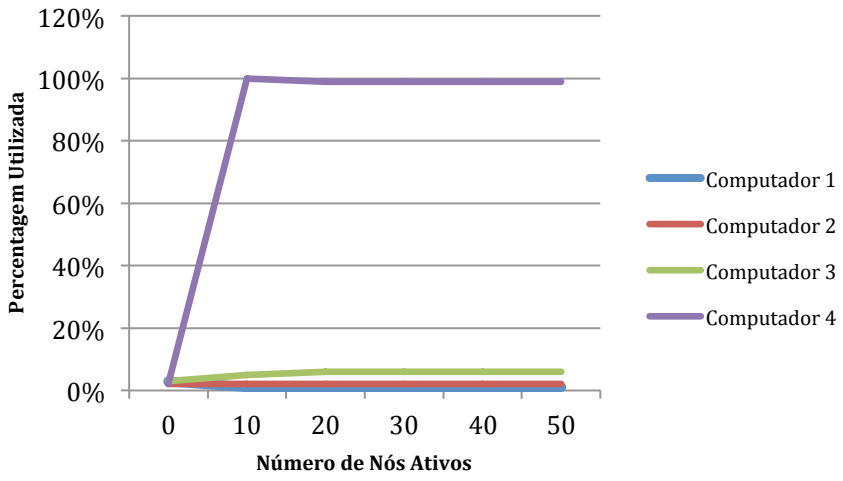

Figura 5.9 - Gráfico comparativo da utilização do processador

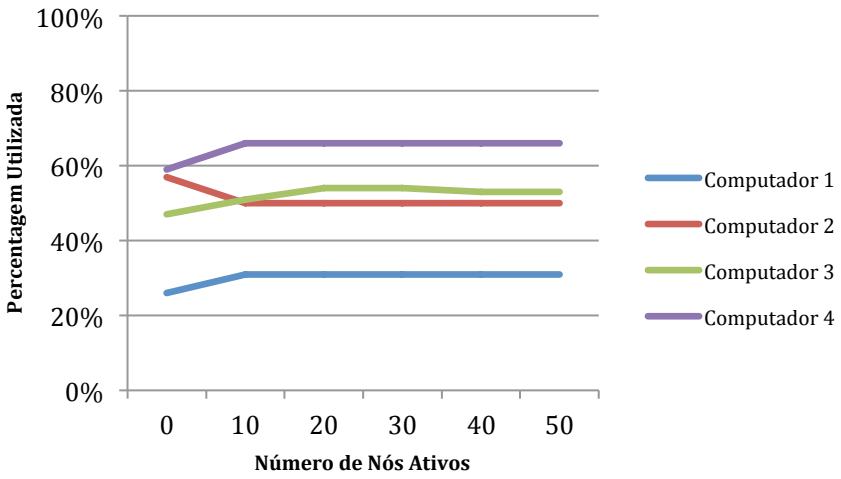

Figura 5.10 - Gráfico comparativo da utilização da memória RAM

Numa segunda fase deste teste, foi trocada a posição do computador 1 com o computador 4 de forma a ser possível testar a estabilidade do computador 4 como *LocalCoordinator* e também para verificar a estabilidade conseguida pelo computador 1 quando albergava o *GlobalCoordinator*.

Sendo esta segunda fase deste teste uma tentativa de explicar a menor estabilidade do computador que alberga o *GlobalCoordinator*, foram feito apenas duas anotações, quando os computadores possuíam 10 e 20 atores associados.

Nº de atores (por *LocalCoordinator*)	Computador 1		Computador 2		Computador 3		Computador 4	
	Processador	Memória RAM	Processador	Memória RAM	Processador	Memória RAM	Processador	Memória RAM
0	3%	26%	2%	57%	3%	47%	2%	59%
10	35%	31%	2%	35%	2%	54%	2%	59%
20	34%	31%	2%	36%	3%	54%	3%	60%

Tabela 5.4 - Percentagem de utilização de processador e memória RAM em função dos nº de nós ativos

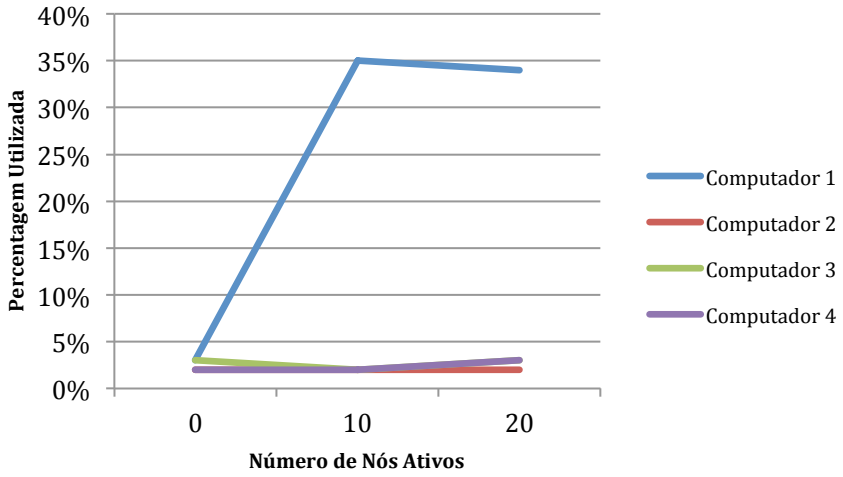

Figura 5.11 - Gráfico comparativo da utilização do processador

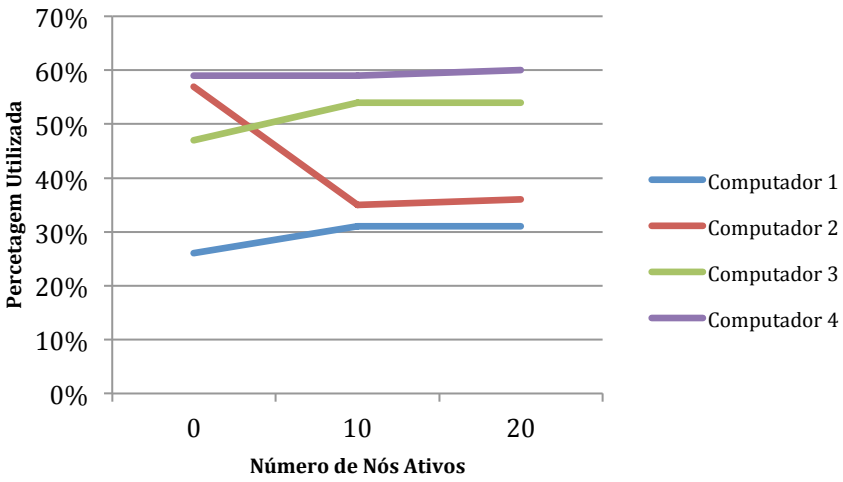

Figura 5.12 - Gráfico comparativo da utilização da memória RAM

5.3.2 Análise de resultados

Como ponto de partida para a análise dos resultados deste teste tem de se assumir os valores que os computadores apresentam quando não tem o simulador em funcionamento, que é o que se pode observar na Tabela 5.2 e na Figura 5.8. É perceptível pelas figuras que todos os computadores têm um estado inicial semelhante, com baixa utilização dos processadores e uma ocupação média da memória RAM. A exceção é o computador 1 que apresenta uma percentagem de ocupação de memória bastante abaixo dos 50%. Se considerarmos que um sistema pode ficar pouco estável a partir de 50% de utilização do seu processa-

dor e 80% da sua memória RAM, todos os computadores estão numa situação perfeitamente estável.

Assim, tendo em conta o número de processos que poderá albergar, é previsível que todos os computadores que hospedam o *LocalCoordinator* vão subindo na percentagem de utilização de processador e da memória. Por outro lado, é esperado que o computador que alberga o *GlobalCoordinator* se mantenha com uma carga aproximadamente constante ao longo do tempo, sendo que essa carga se situará num nível médio, pois durante o normal funcionamento da simulação o *GlobalCoordinator* estará ocupado apenas com a geração de novos atores.

Partindo dos pressupostos assumidos anteriormente, aquando da realização da primeira fase do teste, todos os computadores que albergavam *LocalCoordinators* mantiveram-se estáveis, ao contrário do que aconteceu com o computador que albergava o *GlobalCoordinator* (Tabela 5.3, Figura 5.9 e Figura 5.10), isto porque as percentagens dos computadores que continham os *LocalCoordinators* se mantiveram baixas e a do *GlobalCoordinator* subiu para o limite da carga do processador.

Ora, o que aconteceu nesta primeira fase do teste não foi de encontro com o que era esperado. Uma possível justificação para isto ter acontecido é o facto de os *Actors* usados nos testes, embora tenham uma nomenclatura diferenciada, todos eles se movimentam de uma forma aleatória dentro de uma área restrita. Assim, todos os atores têm uma complexidade de movimento muito baixa e um tempo de processamento também ele muito baixo.

Por outro lado, o *GlobalCoordinator* envolve vários processos que, além de mais complexos do que este tipo de atores, estão em funcionamento constante, o que não acontece com os atores, que a cada movimentação fazem uma pausa de 500 milésimos de segundo.

Para se comprovar a explicação apresentada, foi feita uma segunda fase de testes em que se alterou o computador que hospeda o *GlobalCoordinator*, passando do computador 4 para o computador 1. A escolha do computador 1 para albergar o *GlobalCoordinator* foi suportada no facto de este ser o computador com recursos mais recentes e mais elevados. Deste modo ficaria mais fácil perceber a influência do *GlobalCoordinator* na estabilidade dos computadores.

Efetuada a segunda fase de testes foi possível observar que também o computador 1 aumentou em muito a sua carga (para 35% da carga do processador), não deixando, no en-

tanto, de se manter estável (Tabela 5.4, Figura 5.11 e Figura 5.12). Foi possível assim entender que o *GlobalCoordinator* provoca um aumento de carga do computador em que está albergado, independentemente do sistema operativo e dos componentes do computador. Por outro lado, o computador 4, quando alberga um dos *LocalCoordinators*, tem um comportamento semelhante ao dos outros computadores (Tabela 5.4, Figura 5.11 e Figura 5.12).

Comparando o desempenho do computador 1 e do computador 4 quando albergam o *GlobalCoordinator* (Tabela 5.5, Figura 5.13 e Figura 5.14) é possível verificar que existe uma grande diferença na percentagem de utilização do processador, pois o computador 1 tem uma percentagem à volta dos 35%, enquanto o computador 4 tem uma percentagem de 99% de utilização. O factor decisivo para que exista uma diferença tão grande de percentagem de processador utilizado é os processadores serem de uma série diferente, podendo o processador do computador 1 ser até 4 vezes mais rápido do que o presente no computador 4.

Nº de atores (por *LocalCoordinator*)	Computador 1		Computador 4	
	Processador	Memória RAM	Processador	Memória RAM
0	3%	26%	2%	57%
10	35%	31%	2%	35%
20	34%	31%	2%	36%

Tabela 5.5 - Percentagem de utilização de processador e memória RAM dos 2 computadores que albergaram o *GlobalCoordinator*

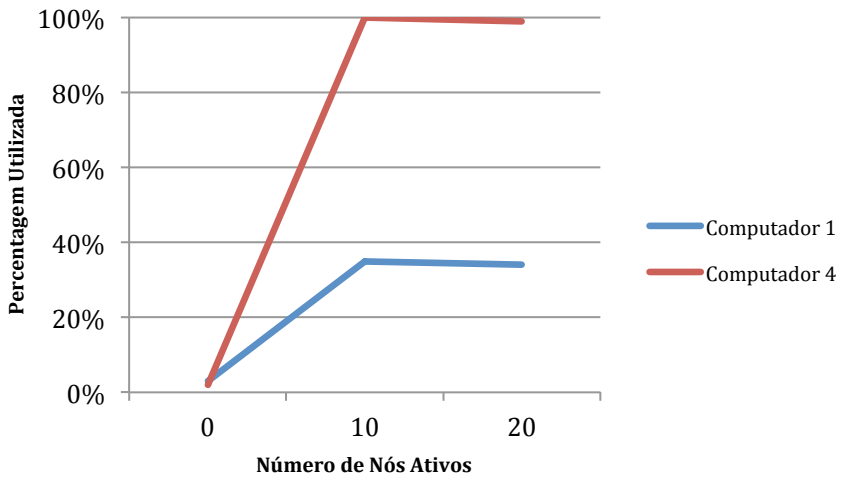

Figura 5.13 - Comparação da utilização do processador pelos 2 computadores que albergaram o *GlobalCoordinator*

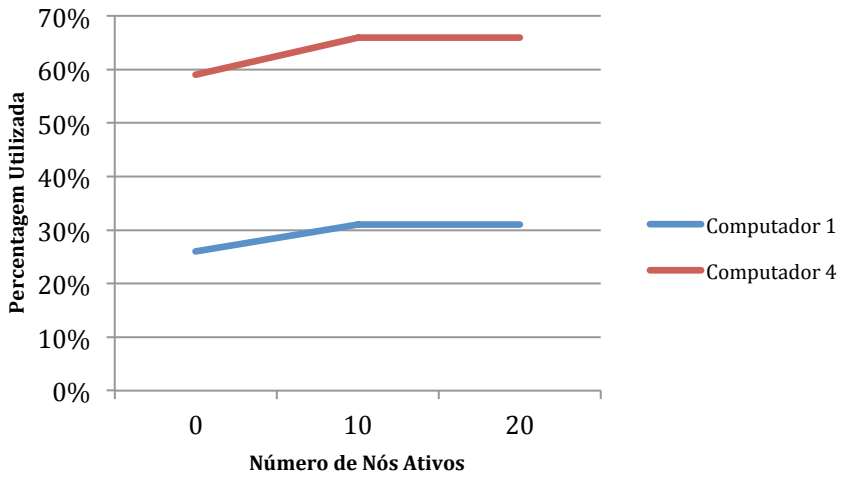

Figura 5.14 - Comparação da utilização da memória RAM pelos 2 computadores que albergaram o *GlobalCoordinator*

5.4 Carga dos processadores

Ao longo de uma simulação o número de atores activos irá estar sempre a aumentar até se atingir um estado estacionário. Assim, se não for inserido qualquer novo *LocalCoordinator*, os novos *Actors* que vão sendo criados terão de ser associados aos *LocalCoordinators* já existentes.

Com este aumento constante do número de nós no sistema é normal que aconteça um aumento da carga sobre os processadores dos computadores, uma vez que cada novo *Actor* é um novo processo. Durante este teste vão ser adicionados *Actors* aos *LocalCoordinators* e verificar a resposta dos processadores ao longo do tempo.

Assim, pretende-se que seja estudada a evolução da carga dos processadores nos diversos computadores em função do número de atores que têm associados a si, de forma a tentar saber qual a influência do número de *Actors* nos processos do simulador e do sistema operativo.

5.4.1 Resultados

Para se perceber a evolução da carga dos processadores ao longo do tempo foi necessário fazer um levantamento da carga dos mesmos em vários instantes de uma simulação. De forma a não tornar a análise do teste demasiado maçadora, os resultados serão apresentados através de uma tabela e um gráfico, baseado na informação apresentada na tabela.

Para este teste o *GlobalCoordinator* foi albergado no computador 1 visto ser previsível que seja o que apresenta melhores condições para isso.

Nº de atores (por *LocalCoordinator*)	Carga do Processador			
	Computador 1	Computador 2	Computador3	Computador 4
10	33%	3%	3%	3%
20	34%	2%	3%	5%
30	36%	2%	2%	3%
40	35%	2%	4%	7%
50	35%	2%	7%	4%
60	36%	2%	5%	5%
70	33%	3%	4%	4%
80	34%	3%	5%	4%
90	32%	4%	5%	5%
100	34%	4%	6%	6%
110	35%	4%	6%	9%
120	35%	4%	6%	5%
130	35%	5%	7%	7%
140	35%	4%	6%	8%
150	36%	4%	6%	7%
160	34%	5%	8%	9%
170	33%	6%	10%	8%
180	32%	6%	12%	10%
190	35%	6%	9%	10%
200	38%	9%	10%	8%
210	37%	8%	9%	11%
220	35%	9%	14%	13%
230	36%	10%	11%	13%
240	36%	9%	15%	15%
250	30%	10%	16%	14%

Tabela 5.6 - Carga do processador em função do número de atores associados

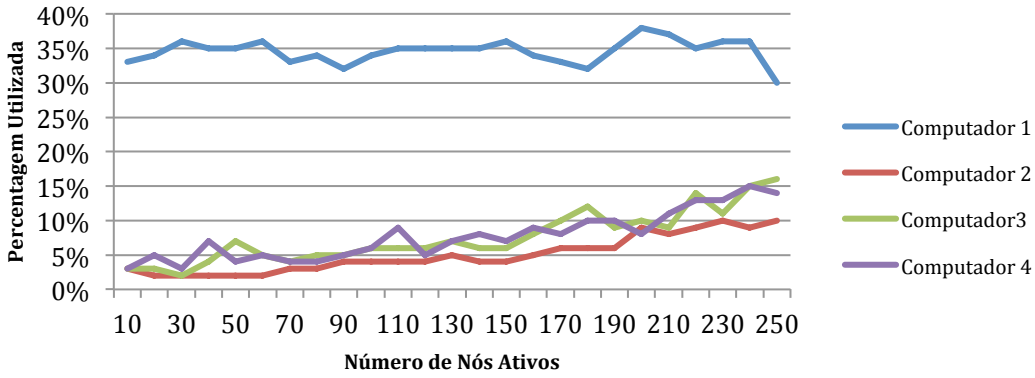

Figura 5.15 - Gráfico comparativo da evolução da utilização do processador

5.4.2 Análise de resultados

Com este teste pretendia-se ver como era a evolução da carga dos processadores dos *LocalCoordinators* à medida que novos atores são adicionados. Perante os resultados obtidos no teste anterior, era de esperar que a carga do simulador do *GlobalCoordinator* se mantivesse constante, compreendia entre os 30% e 40%, e a carga dos *LocalCoordinators* fosse evoluindo, muito lentamente, à medida que vai aumentando o número de atores associados a cada um deles.

Para se ter uma ideia de como os valores evoluem, foram registados na Tabela 5.6 os valores da carga no processador de cada um dos computadores, com um intervalo de 10 atores por amostra, até a um máximo de 250 atores, por *LocalCoordinator*. Após o registo, em tabela, dos dados, foi desenhado um gráfico para que seja mais facilmente perceptível a sua evolução temporal.

Analisando a informação que consta na Figura 5.15 é perceptível que os resultados são os que se esperavam. O computador 1, sendo o *GlobalCoordinator*, apresenta valores de carga entre os 30% e 40%, ou seja, não sofre um aumento percentual com o aumento do número de atores. Já os restantes computadores, funcionando como *LocalCoordinator*, vão aumentando, ainda que brandamente, a percentagem de uso do seu processador à medida que se vão aumentando o número de atores.

É também observável pela Figura 5.15 que a evolução da carga dos processadores dos computadores 2, 3 e 4, ou seja, dos que albergam os *LocalCoordinators*, é semelhante. Quer isto dizer que embora o computador 3 e o computador 4 apresentem valores mais altos que o computador 2, a altura em que o valor do processador sobe é semelhante. Isto pode

levar-nos a concluir que o *hardware*/*software* influência o desempenho dos computadores, mas não os pontos de subida de carga.

5.5 Carga crítica

Como já foi dito, se não forem associados novos *LocalCoordinators* durante uma simulação, inicialmente o número de *Actors* associados a cada *LocalCoordinator* vai estar sempre a aumentar. Durante este teste não serão adicionados novos *LocalCoordinators* à simulação deixando que o inicial chegue a um ponto em que a sua carga seja crítica e já não seja possível que todos os atores sejam processados dentro do tempo esperado.

O objetivo a ser atingido com este teste é o de encontrar o ponto em que a carga se torna crítica em cada um dos *LocalCoordinators* demonstrando que em diferentes máquinas isso acontece em diferentes alturas, dependendo das suas características. Este teste foi feito apenas à carga do processador pois, como se pode ver pelo teste anterior, a memória RAM não sofre grandes alterações com a variação do número de *Actors* presentes na simulação.

5.5.1 Resultados

Na realização deste teste achou-se mais indicado que os resultados fossem apresentados em forma de tabela e gráfico.

De forma a que todos os computadores fossem testados numa situação crítica, o *GlobalCoordinator* foi colocado no computador 1 para ser feito o teste aos computadores 2, 3 e 4, sendo posteriormente colocado no computador 4 para que fosse possível testar o computador 1.

Os valores, em percentagem, que foram considerados como críticos para os processadores dos computadores foram os 50% e 75%, tendo sido anotado o número de atores na altura que os computadores atingiram ou passaram pela primeira vez esse valor.

Carga do Processador	Computador 1	Computador 2	Computador 3	Computador 4
50%	+5000 atores	≈1740 atores	≈1000 atores	≈940 atores
75%	-	≈3300 atores	≈1560 atores	≈1500 atores

Tabela 5.7 - Número de atores associados quando a carga do processador está a 50% e 75%

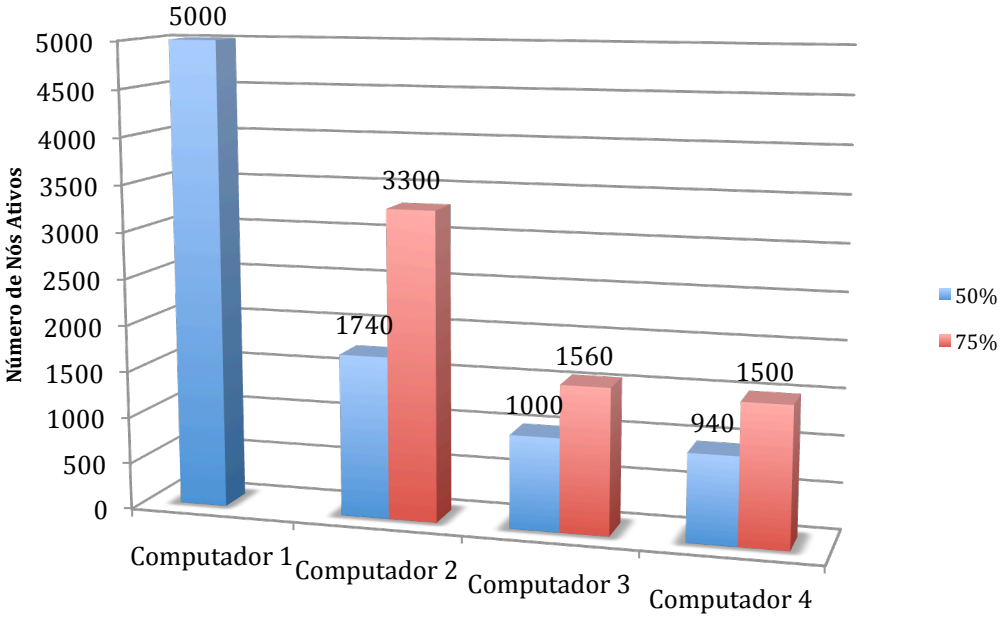

Figura 5.16 - Número de atores associados em percentagens críticas do processador

5.5.2 Análise de resultados

Na abordagem a este teste, teve de ser considerado um valor a partir do qual a carga do processador passava a ser crítica. Embora um processador a 50% da sua carga não seja propriamente crítico, esse valor foi considerado como ponto de referência. Já o valor de 75% da carga achou-se que seria uma boa meta, uma vez que é um valor já significativo de ocupação e que em alguns caso pode afetar o normal funcionamento do computador.

Tal como era esperado, os computadores com menos recursos apresentaram pontos considerados críticos mais rapidamente que os outros. O computador 3 e o computador 4, que são o caso dos computadores em que os seus componentes eram mais fracos e talvez o sistema operativo não fosse o mais indicado, apresentaram valores máximos abaixo dos valores em que o computador 2 atingiu os 50% de carga.

Por outro lado, o computador 2 apresentou valores de atores associados bastante aceitáveis, tendo em conta que todos os atores tinham um movimento totalmente *random*, atingindo um máximo de aproximadamente 3300 atores ativos.

O caso mais evidente de influência dos componentes na capacidade de albergar atores foi o computador 1. No caso deste computador, já levava mais de 5000 atores sem que a carga do seu processador estivesse nos 50% de ocupação. O teste foi interrompido quando este computador já ia com mais de 7000 atores e a percentagem de processador utilizado se

mantilha no valor máximo de 27%, igual ao que tinha com os 5000 atores. Este facto leva-nos a pensar que possivelmente alguma coisa estaria a matar atores. Contudo, estes valores permitem ter uma ideia do que é possível albergar com um computador com componentes de gama alta.

5.6 The ONE

Visto que numa fase inicial deste projeto foi estudado um simulador de ambientes urbanos com algumas semelhanças com o que foi desenvolvido, foi feito um teste informativos sobre a performance do *The ONE*.

5.6.1 Resultados

Para a realização deste teste foi iniciado o simulador com diferentes números de atores e anotada a percentagem de processador utilizada. Este teste foi realizado no computador 4.

Número de Atores	Computador 4
120	55% - 60%
305	60% - 65%
1505	85% - 90%
3005	87% - 92%

Tabela 5.8 - Utilização do processador pelo simulador The ONE

5.6.2 Análise de resultados

Nos testes realizados com o *The ONE* foi sempre utilizado cenário padrão disponibilizados juntamente com o pacote do simulador.

Embora seja possível amentar o número de atores, é perceptível que a percentagem de utilização o processador vai amentando, como era esperado.

No entanto, quando o número de atores aumenta muito dá a sensação de que os nós demoram muito tempo a fazer o seu percurso, chegando por vezes a parar durante certos períodos de tempo.

6 Conclusão

A simulação de redes móveis em ambientes urbanos começa a ter uma importância cada vez maior no mundo dos simuladores. Com o trabalho realizado pretende-se que seja desenvolvido um simulador capaz de simular ambientes particulares, como é o caso das cidades. Um simulador com as características definidas será um grande auxílio para melhor se compreender como funcionam os ambientes citadinos.

Para a realização deste trabalho teve de ser efectuado um estudo sobre alguns simuladores já existentes e também sobre modelos de mobilidade que poderiam ser importantes para a implementação do simulador. Este estudo foi de grande importância visto que permitiu o conhecimento prévio de assuntos relacionados com a simulação de mobilidade de atores.

O trabalho realizado será a base de um simulador que estará em desenvolvimento contínuo, com a possível inclusão de novos tipos de atores e funcionalidades entre atores. Assim, o núcleo deste simulador encontra-se finalizado e preparado para servir de suporte para tudo o que será implementado sobre ele.

Através do trabalho apresentado, é possível verificar que a funcionalidade do núcleo do sistema foi implementada da forma pretendida, pois as entidades são capazes de comunicar entre si, a memória é corretamente partilhada entre todos os *LocalCoordinators* e o *GlobalCoordinator*, a distribuição de novos atores pelo *GlobalCoordinator* é realizada de forma correta, os *Generator*'s são criados e executam ordens de criação de atores de forma correta, e os *LocalCoordinators* conseguem criar novos atores sempre que recebem uma mensagem para tal.

De forma a dotar as simulações de um grande número de atores, o *GlobalCoordinator* é capaz de fazer uma distribuição da carga pelos diversos *LocalCoordinators* associados à simulação, fazendo com que a carga em cada um dos *LocalCoordinators* seja sempre semelhante. Esta distribuição é realizada em função do número de nós que cada *LocalCoordinator* tem associados a si.

Quando se implementa um simulador que funciona sobre um sistema operativo, ou seja, o computador não esta dedicado apenas à simulação, é aconselhado que ele mantenha o computador numa situação estável ao longo do seu funcionamento. Este facto foi tido em conta, tendo sido apresentados testes sobre a estabilidade do simulador e dos computadores que foram inseridos. Através destes testes pode-se concluir que o *GlobalCoordinator* apresenta uma utilização de processador superior ao *LocalCoordinator*. Este resultado não era esperado mas pode ser considerado aceitável se se tiver em conta que os atores não são, ainda, muito desenvolvidos e, por isso, possuem uma taxa de utilização do processador muito reduzida. Assim, é de esperar que com a evolução ao nível dos atores a carga dos processadores dos *LocalCoordinators* se eleve para níveis bastantes superiores ao do *GlobalCoordinator*.

Para que fosse possível ter uma ideia de como estava a arquitetura do simulador, foi efectuado um teste de carga de forma a ser possível estimar qual será o limite de atores que os computadores serão capazes de albergar quando funcionam como *LocalCoordinator*. No que diz respeito a este teste, concluiu-se, tal como se esperava, que quanto melhor forem os componentes do computador, maior é o número de actores que este consegue albergar. Com a realização deste teste foi possível compreender que no sistema operativo *Windows 7* existe uma possível barreira de número de processos que são criados pelo utilizador, pelo que limitou os testes de carga no computador 4 a um número de atores ligeiramente superior a 5000. Isto foi assumido uma vez que foram efetuados diversos testes nos quais os valores do processador usado se mantinham inalterados a partir dos 5000, tendo-se chegado a valores superiores a 7000 atores. No entanto, este teste permitiu saber que o simulador consegue albergar mais de 7000 atores sem nunca ter quebras no seu funcionamento.

Como trabalho futuro, propõe-se que seja feita uma investigação sobre o que obriga a uma utilização tão intensa do processador por parte do *GlobalCoordinator*, fazendo as alterações possíveis para que este não utilize tantos recursos. No que diz respeito ao *GlobalStatus* é possível melhorar o número de variáveis globais existentes, podendo, possivelmente, ser agrupadas num único objeto. Ainda no *GlobalStatus*, poderão ser feitas alterações das funções existentes de forma a torná-lo o máximo possível numa entidade que apenas mantém a informação da simulação. Quanto a melhoramentos futuros, um deles poderá ser os mapas usados nas simulações, uma vez que é expectável que com a evolução dos atores exista a necessidade dos mapas conterem uma quantidade de informação superior àquela

que é utilizada actualmente sob a forma de linhas e pontos. Outro melhoramento possível é a mudança da forma de distribuição de carga pelo número de nós para uma distribuição em função da carga do processador. Por fim, de forma a terminar a construção do simulador, poderá ser finalizado o *reporting* de toda a simulação, finalizar a interface gráfica onde seja possível visualizar o que acontece na simulação e melhorar os vários tipos de atores de forma a que fiquem cada vez mais especializados num tipo de movimento.

Referências

An Introduction to Cooja. Outubro 2010. http://www.sics.se/contiki/wiki/index.php/An_Introduction_to_Cooja (accessed Outubro 2010).

Aschenbruck, Nils, Elmar Gerhards-Padilla, and Peter Martini. "A survey on mobility models for performance analysis in tactical mobile networks." *Journal of Telecommunications and Information Technology*, 2008: 54-61.

Buruhanudeen, Shafinaz, Mohamed Othman, Mazliza Othman, and Borhanuddin Mohd Ali. "Mobility Models, Broadcasting Methods and Factors Contributing Towards the Efficiency of the MANET Routing Protocols: Overview." *IEEE International Conference on Telecommunications and Malaysia International Conference on Communications.* Penang, 2007.

Bai, Fan, and Ahmed Helmy. "Chapter 1 A SURVEY OF MOBILITY MODELS in Wireless Adhoc Networks." In *Wireless Ad-Hoc Networks*, 30. Kluwer Academic Publishes.

Bai, Fan, Narayanan Sadagopan, and Ahmed Helmy. "IMPORTANT: A framework to systematically analyze the Impact of Mobility on Performance of RouTing protocols for Adhoc NeTworks." *Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies.* 2003. 825-835.

Bergamo, M., R.R. Hain, K. Kasera, D. Li, R. Ramanathan, and M. Steenstrup. "System design specification for mobile multimedia wireless network(MMWN) (draft)." Technical report, DARPA project DAAB07-95-C-D156, 1996.

Camp, Tracy, Jeff Boleng, and Vanessa Davies. "A Survery of Mobility Models for Ad Hoc Network Research." *Wireless Communication & Mobile Computing* 2 (September 2002): 483-502.

GloMoSim. Outubro 2010. http://pcl.cs.ucla.edu/projects/glomosim/ (accessed Outubro 2010).

Informatik 4:BonnMotion. University of Bonn. Outubro 2010. http://net.cs.uni-bonn.de/wg/cs/applications/bonnmotion/ (accessed Outubro 2010).

Hong, Xiaoyan, Mario Gerla, Guangyu Pei, and Ching-Chuan Chiang. "A Group Mobility Model for Ad Hoc Wireless Networks." *The ACM International Workshop on Modeling and Simulation of Wireless and Mobile Systems (MSWiM).* Los Angeles, 1999.

Jaap, Sven, Marc Bechler, and Lars Wolf. "Evaluation of Routing Protocols for Vehicular Ad Hoc Networks in City Traffic Scenarios." *5th International Conference on Intelligent Transportation Systems Telecommunications (ITST).* 2005.

Keränen, Ari, Jörg Ott, and Teemu Kärkkäinen. "The ONE Simulator for DTN Protocol Evaluation, Department of Communications and Networking." Article, Department of Communications and Networking, Helsinki University of Technology, Helsinki, 2009.

Liang, Ben, and Zygmunt J. Haas. "Predictive Distance-Based Mobility Management for PCS Networks." *Proceedings of IEEE Information Communications Conference (INFOCOM 1999).* 1999.

Musolesi, Mirco, and Cecilia Mascolo. "A Community Based Mobility Model for Ad Hoc Network Research." *2nd ACM/SIGMOBILE Int. Worksh. Multi-hop Ad Hoc Netw. Theory Real. REALMAN'06.* Florence,, 2006. 31-38.

Musolesi, Mirco, and Cecilia Mascolo. "Chapter 1 Mobility Models for Systems Evaluation A Survey." Dartmouth College; University of Cambridge, 28.

MiXiM. Outubro 2010. http://mixim.sourceforge.net/ (accessed Outubro 2010).

OpenJUMP GIS. http://www.openjump.org/ (accessed Outubro 2010).

SUMO - Simulation of Urban Mobility. Outubro 2010. http://sumo.sourceforge.net/ (accessed Outubro 2010).

Sánchez, Miguel, and Pietro Manzoni. "ANEJOS: a Java based simulator for ad hoc networks." *Future Generation Computer Systems* 17 (2001): 573–583.

Ribeiro, Andrea, and Rute C. Sofia. "A Survey of Mobility Models on Wireless Networks." SITI Technical, Lusífona University, Lisboa, 2011, 13.

The Network Simulator - ns-2. Outubro 2010. http://isi.edu/nsnam/ns/ (accessed Outubro 2010).

The ONE. Outubro 2010. http://www.netlab.tkk.fi/tutkimus/dtn/theone/ (accessed Outubro 2010).